# Java Map Interface Implementation Classes

## HashMap in Java | Methods, Use, Example

**HashMap in Java** is an unordered collection that stores elements (objects) in the form of key-value pairs (called entries).
It is expressed as HashMap<Key, Value>, or HashMap<K, V>, where K stands for key and V for value. Both Key and value are objects. HashMap uses an object to retrieve another object.

If the key is provided, its associated value can be easily retrieved from the HashMap. Keys in the map must be unique which means we cannot use duplicate data for keys in the HashMap.

If we try to insert an entry that has a duplicate key, the map replaces the old entry with a new entry.

Java HashMap class is one of four concrete implementations of the Map interface. It was added in Java 1.2 version. It is found in java. util.HashMap<K, V> package. It is efficient for locating a value, adding an entry, and deleting an entry.

### Hierarchy of HashMap class in Java

HashMap class in Java extends AbstractMap class and implements the Map interface. The hierarchy diagram of HashMap can be seen in the below figure.

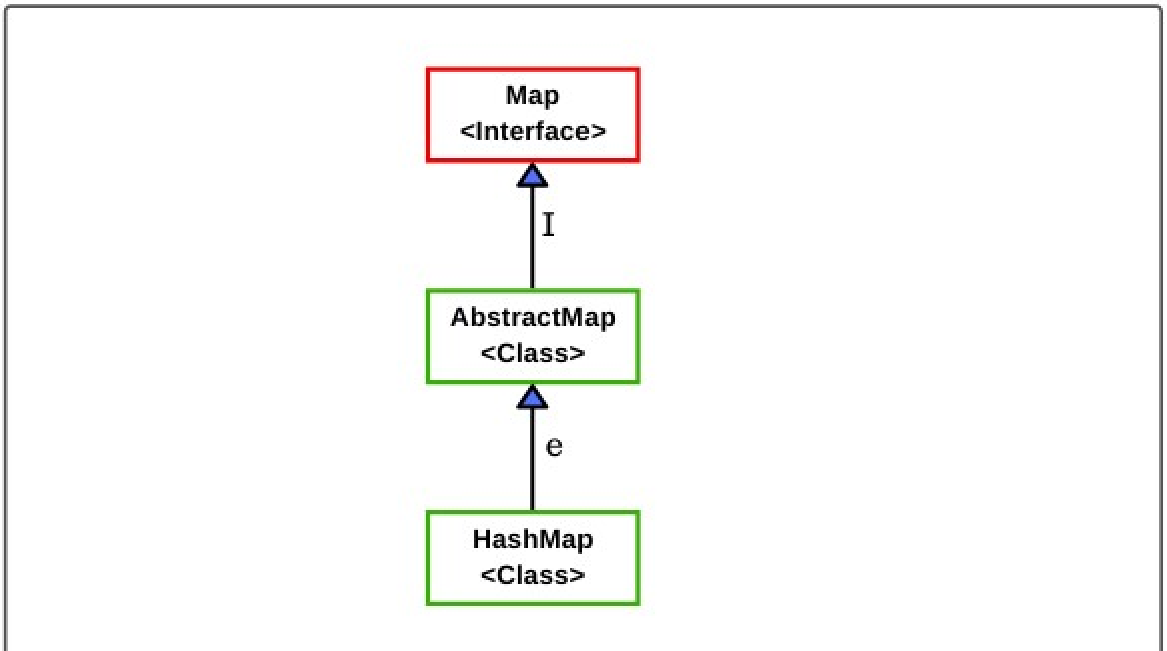


Fig: Java HashMap Hierarchy Diagram

## HashMap class Declaration

---

HashMap is a generic class that can be declared as follows:

```
public class HashMap<K,V>

  extends AbstractMap<K,V>

  implements Map<K,V>, Cloneable, Serializable
```

## Features of Java HashMap class

---

# Java Map Interface Implementation Classes

There are several features of HashMap class that need to keep in mind. They are as:

1. The underlying data structure of HashMap is HashTable. In simple words, HashMap internally uses hash table for storing entries. That means accessing and adding an entry almost as fast as accessing an array.
2. In HashMap, insertion order is not preserved (i.e. maintains no order). Which means we cannot retrieve keys and values in the same order in which they have been inserted into the HashMap.

3. It is based on the Hashcode of keys, not on the hash code of values.

4. Java HashMap contains only unique keys that means no duplicate keys are allowed but values can be duplicated. We retrieve values based on the key.

5. Heterogeneous objects are allowed for both keys and values.

6. Java HashMap can have only one null key because duplicate keys are not allowed.

7. Multiple null values are allowed in the HashMap.

8. HashMap in Java is not synchronized that means while using multiple threads on the HashMap object, we will get unreliable results.

9. Java HashMap implements Cloneable, and Serializable interfaces but not implements Random Access.
10. HashMap is the best choice if our frequent operation is a search operation.

11. If no element exists in the HashMap, it will throw an exception named **NoSuchElementException**.
12. Java HashMap stores only object references. Therefore, we cannot use primitive data types like double or int. We can use wrapper class like Integer or Double instead.

# Java Map Interface Implementation Classes

## Constructors of HashMap class

HashMap class in Java provides four constructors that are as follows:
**1. HashMap():** It is used to construct an empty HashMap object with the default initial capacity of 16 and the default fill ratio (load factor) is 0.75. The syntax to create a hash map object is as follows:

```
HashMap hmap = new HashMap();

HashMap<K, V> hmap = new HashMap<K,V>(); // Generic form.
```

**2. HashMap(int initialCapacity):** It is used to create an empty hash map object with a specified initial capacity under the default load factor 0.75. The general syntax is as follows:

```
HashMap<K,V> hmap = new HashMap<K,V>(int initialCapacity);
```

**3. HashMap(Map m):** This constructor is used to create hash map object by initializing the elements of the given Map object m.
**4. HashMap(int initialCapacity, float loadFactor):** This constructor is used to create hash map object with specified initial capacity and load factor.
The general syntax to create hash map object with initial capacity and load factor is as:

```
HashMap<K,V> hmap = new HashMap<>(int initialCapacity, float loadFactor);



For example:

    HashMap<String, Integer> hmap = new HashMap<>(30, 0.7f);
```

In the above syntax, capacity is the number of buckets in which hash map values can be stored. The load factor is the amount of buckets that are used before the capacity automatically is grown.

The value is a floating-point number that has ranges from 0 (empty) to 1.0 (full). 0.7 means when buckets are 70% full, the capacity is increased. The default capacity is 16 and the load factor is 0.75 that is often sufficient.

## HashMap Methods in Java

---

In this section, we have listed several important methods available in HashMap class that are as follows:

1. **void clear():** It is used to remove entries from the specified map.
2. **boolean isEmpty():** This method is used to check whether the map is empty or not. If there are no key-value mapping present in the hash map then it returns true, else false.
3. **Object clone():** It is used to create a shallow copy of this HashMap. Only hashmap and all entries are cloned, not elements. Both this map and new map share references to the same keys and values.

4. **Set entrySet():** It is used to return a set view containing all of the key/value pairs in this map.
5. **Set keySet():** This method is used to retrieve a set view of the keys in this map.
6. **V put(Object key, Object value):** This method is used to insert an entry in the map.
7. **void putAll(Map map):** This method is used to insert all the entries of the specified map to another map.
8. **V putIfAbsent(K key, V value):** This method adds the specified value associated with the specified key in the map only if it is not already specified.
9. **V remove(Object key):** This method is used to delete an entry for the specified key.

10. **boolean remove(Object key, Object value):** This method removes the specified value associated with specific key from the map.
11. **int size():** This method returns the number of entries in the map.
12. **Collection values():** This method returns a collection view containing all of the values in the map.
13. **V get(Object key):** This method is used to retrieve the value associated with a key. Its return type is Object.
14. **V replace(K key, V value):** This method is used to replace the specified value for a specified key.
15. **boolean replace(K key, V oldValue, V newValue):** This method is used to replace the old value with the new value for a specified key.
16. **boolean containsValue(Object v):** This method is used to determine if map contains a particular value. It returns true if some value is equal to the v.
17. **boolean containsKey(Object k):** This method is used to determine if the map contains a particular key. It will return true if some key in this map is equal to k.
18. **boolean equals(Object o):** This method is used to compare the specified Object with the Map.

## Java HashMap Example Programs

---

1. Let's create a program where we will simply add entries in the HashMap and display it on the console.

**Program source code 1:**

```
import java.util.HashMap;

public class HashMapEx1 {

public static void main(String[] args)

{
```

```
// Create a HashMap.
  HashMap<String,Integer> hmap = new HashMap<>();


// Checking HashMap is empty or not.
   boolean empty = hmap.isEmpty();
   System.out.println("Is HashMap empty: " +empty);


// Adding entries in the hash map.
   hmap.put("John", 24); // hmap.size() is 1.
   hmap.put("Deep", 22); // hmap.size() is 2.
   hmap.put("Shubh", 15); // hmap.size() is 3.
   hmap.put("Riky", 22); // hmap.size() is 4. // Adding duplicate value.
   hmap.put("Mark", 30); // hmap.size() is 5.


System.out.println("Entries in HashMap: " +hmap);
int size = hmap.size();
System.out.println("Size of HashMap: " +size);


// Adding null key and value.
   hmap.put(null, null); // hmap.size() is 6.
  System.out.println("Updated entries in HashMap: " +hmap);

  }

}

Output:

    Is HashMap empty: true

    Entries in HashMap: {Riky=22, Shubh=15, John=24, Mark=30, Deep=22}

    Size of HashMap: 5

    Updated entries in HashMap: {null=null, Riky=22, Shubh=15, John=24, Mark=30,
Deep=22}
```

As you can see from the above output, the entries in the HashMap are in no particular order in which they are inserted in map. We have stored String as keys, and Integer as values as so we are using HashMap<String, Integer> as the type. The put() method adds the entries to the map.

2. Let's take an example program in which we will try to add duplicate keys and values in HashMap.

**Program source code 2:**

```
import java.util.HashMap;

public class HashMapEx2 {

public static void main(String[] args)

{

 HashMap<Integer, String> hmap = new HashMap<>();



 hmap.put(5, "Banana");
 hmap.put(10, "Mango");

 hmap.put(15, "Apple");


System.out.println("Entries in HashMap: " +hmap);

System.out.println("Size of HashMap: " +hmap.size());


// Adding duplicate key in hash map.

   hmap.put(10, "Guava"); // Still hmap.size is 3.

   hmap.put(20, "Banana"); // Adding duplicate value.


   System.out.println("Updated entries in HashMap: " +hmap);

   System.out.println("Size after adding duplicate value: " +hmap.size());

   }

}

Output:

      Entries in HashMap: {5=Banana, 10=Mango, 15=Apple}

      Size of HashMap: 3

      Updated entries in HashMap: {20=Banana, 5=Banana, 10=Guava, 15=Apple}

      Size after adding duplicate value: 4
```

As you can see in this program, we cannot store duplicate keys in HashMap. However, we have tried to store a duplicate key with another value, but it simply replaced the value.

3. Let's create a program where we will perform the remove operation. We will remove entry from the HashMap using remove() method. Look at the source code.

**Program source code 3:**

```
import java.util.HashMap;

public class HashMapEx3 {

public static void main(String[] args)

{

 HashMap<Character,String> hmap = new HashMap<>();



 hmap.put('R', "Red");

 hmap.put('O', "Orange");

 hmap.put('G', "Green");

 hmap.put('B', "Brown");

 hmap.put('W', "White");



// Displaying HashMap entries.

   System.out.println("Entries in HashMap: " +hmap);
// Removing Key-Value pairs for key 'B'.

  Object removeEntry = hmap.remove('B');

  System.out.println("Removed Entry: " +removeEntry);

  System.out.println("HashMap Entries after remove: " +hmap);



// Checking entry is removed or not.

  Object removeEntry2 = hmap.remove('W', "White");

  System.out.println("Is entry removed: " +removeEntry2);

  System.out.println("Updated HashMap entries: " +hmap);

   }

}

Output:

       Entries in HashMap: {R=Red, B=Brown, G=Green, W=White, O=Orange}

       Removed Entry: Brown

       HashMap Entries after remove: {R=Red, G=Green, W=White, O=Orange}

       Is entry removed: true

       Updated HashMap entries: {R=Red, G=Green, O=Orange}
```

4. Let's take an example program where we will replace a specified value for the specified key. Look at the source code.

**Program source code 4:**

```
import java.util.HashMap;
```

## Java Map Interface Implementation Classes

```
public class HashMapEx4 {

public static void main(String[] args)

{

 HashMap<Character,String> hmap = new HashMap<>();



 hmap.put('R', "Red");

 hmap.put('O', "Orange");

 hmap.put('G', "Green");

 hmap.put('B', "Brown");

 hmap.put('W', "White");



// Displaying HashMap entries.

   System.out.println("Entries in HashMap: " +hmap);



// Replacing specified value for the specified key.

   Object replaceValue = hmap.replace('B', "Black");

   System.out.println("Replaced value: " +replaceValue);

   System.out.println("Updated entries in HashMap: " +hmap);



  boolean replaceValue2 = hmap.replace('G', "Green", "Greenish");
  System.out.println("Is value replaced: " +replaceValue2);

  System.out.println(hmap);

   }

}

Output:

       Entries in HashMap: {R=Red, B=Brown, G=Green, W=White, O=Orange}

       Replaced value: Brown

       Updated entries in HashMap: {R=Red, B=Black, G=Green, W=White, O=Orange}

       Is value replaced: true

       {R=Red, B=Black, G=Greenish, W=White, O=Orange}
```

## Use of HashMap in Java

Java HashMap can be the best choice if we want to perform a search operation. It is designed to rapidly find things. The best example of this kind is phonebook. The name of person (a string) can be used to search the person's phone number.

Let's understand it with a simple example program. Look at the following source code.

**Program source code 5:**

```
import java.util.HashMap;

public class HashMapUse {

public static void main(String[] args)
{
 HashMap<String, Long> hmap = new HashMap<>();


 hmap.put("John", 9431676282L);

 hmap.put("Deep", 8292736478L);

 hmap.put("Shubh", 8123543268L);

 hmap.put("Mark", 9876789142L);

 hmap.put("Ricky", 8768976872L);


// Retrieve number with its key by calling get() method.

   Long number = hmap.get("Deep");

   System.out.println("Deep's phone number: " +number);


   Long number2 = hmap.getOrDefault("Steave", -1L);

   System.out.println("Alex's phone number: " +number2);

   }

}

Output:

      Deep's phone number: 8292736478

      Alex's phone number: -1
```

## Java Map Interface Implementation Classes

In this example, we have created a HashMap called phonebook with keys (person's name) that are strings and values (person's phone number) that are Long objects. Objects are stored in the hash map by calling put(Object key, Object value) method.

This method stores an item on the map with key as name and value as a phone number. hmap.put("John", 9431676282L);. If the specified key is not found, -1 is returned by default.

## How to Iterate HashMap in Java Example

In this tutorial, we will learn how to **iterate** **HashMap in Java**. Java provides multiple convenient ways to iterate over a hash map.
We can iterate over keys, values, or both. We can also remove an element from a map while iterating over it.

Let's understand the following ways to iterate HashMap in Java.

### Iterating HashMap using Iterator

Let's take an example program where we will iterate hash map in java using java iterator concept. Look at the following source code.
**Program source code 1:**

```
import java.util.HashMap;

import java.util.Iterator;

import java.util.Map.Entry;

public class HashMapIterating1  {

public static void main(String[] args)

{
```

# Java Map Interface Implementation Classes

```
 HashMap<Character, String> hmap = new HashMap<>(); // Creating a hash map.



  hmap.put('V', "Violet");

  hmap.put('I', "Indigo");

  hmap.put('B', "Blue");

  hmap.put('G', "Green");

  hmap.put('Y', "Yellow");

  hmap.put('O', "Orange");

  hmap.put('R', "Red");



Iterator<Entry<Character, String>> itr = hmap.entrySet().iterator(); // entrySet is a
method that is used to get view of entries of a hash map.

System.out.println("Iterating Entries of HashMap");

while(itr.hasNext())

{

 Object key = itr.next();

 System.out.println(key);

}

Iterator<Character> itr2 = hmap.keySet().iterator(); // keySet is a method that is used to
get view of keys of a hash map.

System.out.println("Iterating Keys of HashMap");
```

```
while(itr2.hasNext())

{

 Object keyView = itr2.next();

 System.out.println(keyView);

}

Iterator<String> itr3 = hmap.values().iterator(); // values is a method that is used to get
values of keys of a hash map.

System.out.println("Iterating Values of HashMap");

while(itr3.hasNext())

{

 Object valuesView = itr3.next();

 System.out.println(valuesView);

  }

 }

}
```

Output:

```
      Iterating Entries of HashMap

      B=Blue

      R=Red

      V=Violet
```

```
 G=Green
 I=Indigo
 Y=Yellow
 O=Orange
 Iterating Keys of HashMap
 B
 R
 V
 G
 I
 Y
 O
Iterating Values of HashMap
Blue
Red
Violet
Green
Indigo
Yellow
Orange
```

Let’s create another program where we will remove the last entry of a hash map returned by the Iterator.

**Program source code 2:**

```
import java.util.HashMap;

import java.util.Iterator;

import java.util.Map.Entry;

public class IterationRemove {

public static void main(String[] args)

{

 HashMap<Integer, String> hmap = new HashMap<>();



  hmap.put(5, "Five");

  hmap.put(10, "Ten");

  hmap.put(15, "Fifteen");

  hmap.put(20, "Twenty");

  hmap.put(25, "Twenty-five");

  hmap.put(30, "Thirty");



Iterator<Entry<Integer, String>> itr = hmap.entrySet().iterator(); // entrySet is a method
that is used to get view of entries of a map.
System.out.println("Iterating Entries of HashMap");

while(itr.hasNext())

{

  Object key = itr.next();

  System.out.println(key);

 }

// Removing last entry returned by Iterator

  itr.remove(); // This method will remove last entry of a hash map while iterating.

  System.out.println("Entries of HashMap after removing: " +hmap.entrySet());

 }

}
```

Output:

```
      Iterating Entries of HashMap

      20=Twenty

      5=Five

      25=Twenty-five

      10=Ten

      30=Thirty

      15=Fifteen
```

```
    Entries of HashMap after removing: [20=Twenty, 5=Five, 25=Twenty-five, 10=Ten,
30=Thirty]
```

## Iterating over keys or values using keySet() and value() methods

---

This technique is useful when you want to get a set view of keys or values of a hash map. Using keySet(), values(), and for-each loop, you can iterate over keys or values of a hash map.

Let's take an example program where we will iterate over keys or values of a hash map using keySet() and values() methods.
keySet() method returns a set view of the keys from a hash map and values() method returns a collection-view of values from a hash map.

**Program source code 3:**

```
import java.util.HashMap;

public class HashMapIterating2 {

public static void main(String[] args)

{

 HashMap<Integer, String> hmap = new HashMap<>();



  hmap.put(5, "Five");

  hmap.put(10, "Ten");

  hmap.put(15, "Fifteen");

  hmap.put(20, "Twenty");
  hmap.put(25, "Twenty-five");

  hmap.put(30, "Thirty");



for (Integer num : hmap.keySet()) // Iterating over keys of a hashmap using keySet()
method.

 System.out.println("Number: " +num);

 System.out.println();



for (String word : hmap.values()) // Iterating over values of a hashmap using values()
method.

 System.out.println("Word: " +word);

 }

}
```

Output:

```
      Number: 20

      Number: 5

      Number: 25

      Number: 10

      Number: 30

      Number: 15
```

```
    Word: Twenty

    Word: Five

    Word: Twenty-five

    Word: Ten

    Word: Thirty

    Word: Fifteen
```

## Iterating Java HashMap using Map.Entry<K,V>method

---

Let's create a program where we will iterate entry of a hash map using Map.Entry<K,V> method. We will use the following methods for iteration.

**Program source code 4:**

```
import java.util.HashMap;

import java.util.Map.Entry;

public class HashMapIterating3 {

public static void main(String[] args)

{

 HashMap<Integer, String> hmap = new HashMap<>();



  hmap.put(1012, " John");

  hmap.put(1202, " Ricky");
  hmap.put(1303, " Deep");

  hmap.put(1404, " Mark");

  hmap.put(1505, " Maya");



for (Entry<Integer, String> entry : hmap.entrySet()) // Iterating over entries of a map
using entrySet() method.

{

   System.out.println("Id: " + entry.getKey() + ", Name: " + entry.getValue());

}

 }

}

Output:

      Id: 1505, Name:  Maya

      Id: 1202, Name:  Ricky

      Id: 1012, Name:  John

      Id: 1303, Name:  Deep

      Id: 1404, Name:  Mark
```

## Iterating Java HashMap using forEach() method

# Java Map Interface Implementation Classes

Let's take an example program based on the forEach() method. This is the best efficient and simple way. In this program, we will use lambda expression inside the forEach() method to display each entry of the hash map.

**Program source code 5:**

```
import java.util.HashMap;

public class HashMapIterating4 {

public static void main(String[] args)

{

 HashMap<String, String> hmap = new HashMap<>();



  hmap.put("India", " Delhi");

  hmap.put("USA", " Washington, D.C.");

  hmap.put("Australia", " Canberra");

  hmap.put("New Zealand", " Wellington");

  hmap.put("Switzerland", " Bern");



//Iteration over map using forEach() method.

   hmap.forEach((k,v) -> System.out.println("Country: "+ k + ", Capital: " + v));

 }

}

Output:
```

```
Country: USA, Capital:  Washington, D.C.

Country: New Zealand, Capital:  Wellington

Country: Australia, Capital:  Canberra

Country: Switzerland, Capital:  Bern

Country: India, Capital:  Delhi
```

## Internal Working of HashMap in Java 8

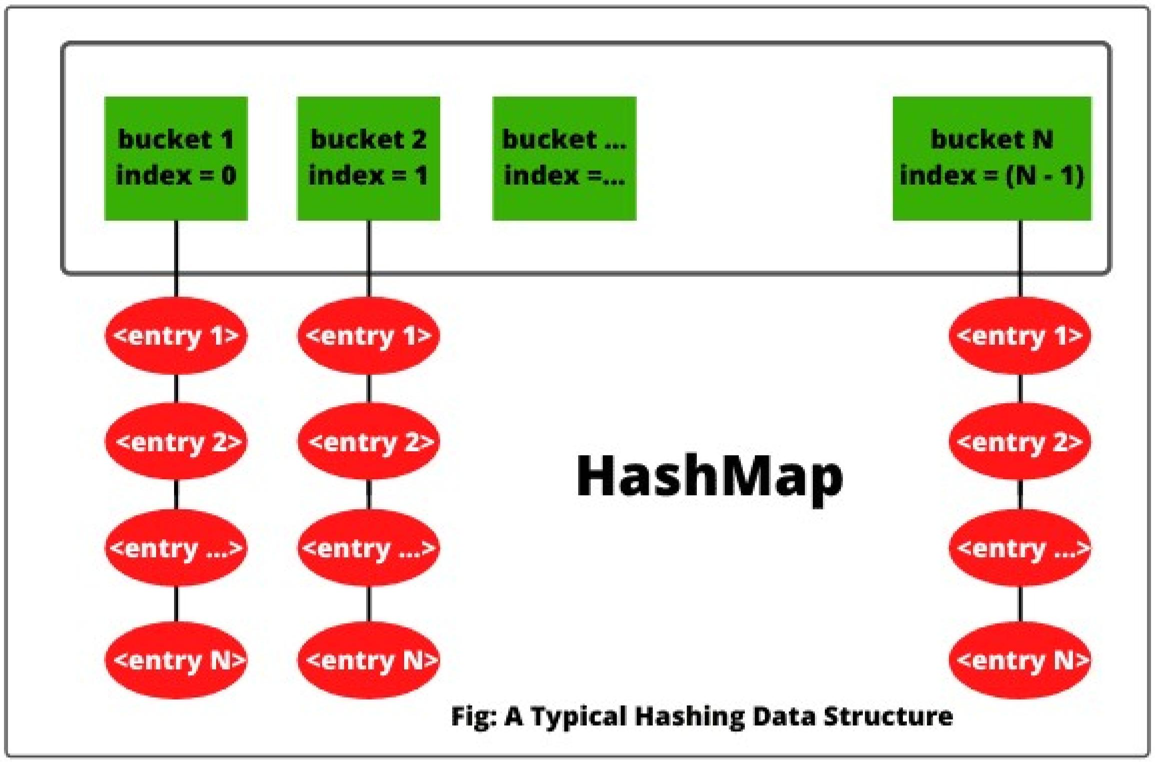
**Internal Working of HashMap in Java |** In this tutorial, we will learn how HashMap works in Java internally step by step with example.
If you do not familiar with the basic features of Java HashMap, first go to this tutorial and then come to learn the internal working of HashMap in Java.
**HashMap in Java** is basically an array of buckets (also known as bucket table of HashMap) where each bucket uses linked list to hold elements. A linked list is a list of nodes where each node contains a key-value pair.
In simple words, a bucket is a linked list of nodes where each node is an object of class Node<K,V>. The key of the node is used to obtain the hash value and this hash value is used to find the bucket from Bucket Table.

HashMap works on the principle of hashing data structure or technique that uses an object's hashcode to place that object inside the map.

Hashing involves Bucket, Hash function (hashCode() method), and Hash value. It provides the best time complexity of O(1) for insertion and retrieval of objects.

Therefore, it is the best-suited data structure for storing key-value pairs that later on can be retrieved in minimum time.

A typical hashing data structure for storing key-value pairs can be seen in the below figure.



## Initial Capacity and Load Factor of Java HashMap

Load Factor and Initial Capacity are two important factors that plays important role in the internal working of HashMap in Java.

**Initial Capacity** is a measure of the number of buckets or size of bucket array internally by HashMap at the time of the creation of HashMap. The default initial capacity of HashMap is 16 (i.e. the number of buckets). It is always expressed in the power of 2 (2, 4, 8, 16, etc) reaching maximum of 1 << 30 (2^30).

**Load Factor** is a factor that is internally used by HashMap to determine when the size of Bucket array requires to be increased. By default, it is 0.75.

When the number of nodes in the HashMap is more than 75% of total capacity, HashMap grows its bucket array size. The capacity of HashMap always doubled each time when HashMap needs to be increased its bucket array size.

**Bucket Table:**
An array of buckets is called bucket table of HashMap. In the bucket table, a bucket is a linked-list of nodes where each node is an object of class Node<K, V>.

The key of node is used to obtain the hash value and this hash value is used to calculate the index of the bucket from the bucket table in which key-value pairs will be placed.

**Node:**
Each node of the linked-list is an object of class Node<K,V>. This class is a static inner class of HashMap class that implements the Map.Entry<K,V> interface.

The general syntax for inner static node class of HashMap is as follows:

```
static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

V value;

Node<K,V> next;

}
```

final int hash: It is a hash value of the key.

final K key: It is a key of the node.

V value: It gives the value of the node.

Node<K,V> next: It indicates the pointer to the next node present in the bucket or linked-list.

## Role of hashCode() and equals() methods in Internal Working of Java HashMap

---

The hashcode() and equals() play a major role in the internal working of HashMap in Java. Therefore, before understanding the internal working of HashMap, you should be aware of basic knowledge of hashCode() and equals() methods.

**hashCode():**
Hash function is a function that maps a key to an index in the hash table. It obtains an index from a key and uses that index to retrieve the value for a key.

A hash function first converts a search key (object) to an integer value (known as hash code) and then compresses the hash code into an index to the hash table.

The Object class (root class) of Java provides a hashCode method that other classes need to override. hashCode() method is used to retrieve the hash code of an object. It returns an integer hash code by default that is a memory address (memory reference) of an object.

The general syntax for the hashCode() method is as follows:

```
public native hashCode()



The general syntax to call hashCode() method is as follows:

  int h = key.hashCode();
```

The value obtained from the hashCode() method is used as the bucket index number. The bucket index number is the address of the entry (element) inside the map. If the key is null then the hash value returned by the hashCode() will be 0.

**equals():**
The equals() method is a method of Object class that is used to check the equality of two objects. HashMap uses equals() method to compares Keys whether they are equal or not.

The equals() method of Object class can be overridden. If we override the equals() method, it is mandatory to override the hashCode() method.

## Put Operation: How put() method of Hashmap works internally in Java?

---

The put() method of HashMap is used to store the key-value pairs. The syntax of put() method to add key/value pair is as follows:

```
hashmap.put(key, value);
```

Let's take an example where we will insert three (Key, Value) pairs in the HashMap.

```
HashMap<String, Integer> hmap = new HashMap<>();



 hmap.put("John", 20);

 hmap.put("Harry", 5);

 hmap.put("Deep", 10);
```

Let's understand at which index the key-value pairs will be stored into HashMap.

When we call the put() method to add a key-value pair to hashmap, HashMap calculates a hash value or hash code of key by calling its hashCode() method. HashMap uses that code to calculate the bucket index in which key/value pair will be placed.

The formula for calculating the index of bucket (where n is the size of an array of the bucket) is given below:

```
Index = hashCode(key) & (n-1);
```

Suppose the hash code value for “John” is 2657860. Then the index value for “John” is:

```
Index = 2657860 & (16-1) = 4
```

The value 4 is the computed index value where the key and value will be store in HashMap.

**Note:** Since HashMap allows only one null Key, the hash value returned by the hashCode(key) method will be 0 because the hashcode for null is always 0. The 0th bucket location will be used to place key/value pair.

## How is Hash Collision occurred and resolved?

---

A hash collision occurs when hashCode() method generates the same index value for two or more keys in the hash table. To overcome this issue, HashMap uses the technique of linked-list.

When hashCode() method produces the same index value for a new Key and the Key that already exists in the hash table, HashMap uses the same bucket index that already contains nodes in the form of linked-list.

A new node is created at the last of the linked-list and connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at the same index value.

When a new value object is inserted with an existing Key, HashMap replaces the old value with the current value related to the Key. To do it, HashMap uses equals() method.

This method check that both Keys are equal or not. If Keys are the same, this method returns true and the value of that node is replaced with the current value.

## How get() method in HashMap works internally in Java?

---

The get() method in HashMap is used to retrieve the value by its key. If we don't know the Key, it will not fetch the value. The syntax for calling get() method is as follows:

```
value = hashmap.get(key);
```

When the get(K Key) method takes a Key, it calculates the index of bucket using the method mentioned above. Then that bucket's List is searched for the given key using equals() method and final result is returned.

## Time Complexity of put() and get() methods

---

HashMap stores a key-value pair in constant time which is O(1) for insertion and retrieval. But in the worst case, it can be O(n) when all node returns the same hash value and inserted into the same bucket.

The traversal cost of n nodes will be O(n) but after the changes made by Java 1.8 version, it can be maximum of O(log n).

## Concept of Rehashing

**Rehashing** is a process that occurs automatically by HashMap when the number of keys in the map reaches the threshold value. The threshold value is calculated as **threshold** = capacity * (load factor of 0.75).
In this case, a new size of bucket array is created with more capacity and all the existing contents are copied over to it.

For example:

```
Load Factor: 0.75

Initial Capacity: 16 (Available Capacity initially)

Threshold = Load Factor * Available Capacity = 0.75 * 16 = 12
```

When 13th key-value pair is inserted into the HashMap, HashMap grows its bucket array size to 16*2 = 32.

```
Now Available capacity: 32



Threshold = Load Factor * Available Capacity = 0.75 * 32 = 24
```

Next time when 25th key-value pair is inserted into HashMap, HashMap grows its bucket array size to 32*2 = 64 and so on.

## LinkedHashMap in Java | Methods, Example

**LinkedHashMap in Java** is a concrete class that is HashTable and LinkedList implementation of Map interface. It stores entries using a doubly-linked list.
Java LinkedHashMap class extends the HashMap class with a linked-list implementation that supports an ordering of the entries in the map.

# Java Map Interface Implementation Classes

LinkedHashMap in Java was added in JDK 1.4 version. It is exactly the same as HashMap (including constructors and methods) except for the following differences:

1. The underlying data structure of HashMap is HashTable whereas, the underlying data structure of LinkedHashMap is HashTable and LinkedList (Hybrid data structure).
2. Insertion order is not preserved in the HashMap because it is based on the hashCode of Key. But in the case of LinkedHashMap, the insertion order of elements is preserved because it is based on the Key insertion order, that is, the order in which keys are inserted in the map.

3. HashMap was introduced in Java 1.2 version whereas, LinkedHashMap was introduced in Java 1.4 version.

## Hierarchy of LinkedHashMap in Java

---

LinkedHashMap implementation in Java is a subclass of HashMap class. That is, LinkedHashMap class extends the HashMap class and implements Map interface.

The hierarchy diagram of LinkedHashMap is shown in the below figure.

## Java Map Interface Implementation Classes

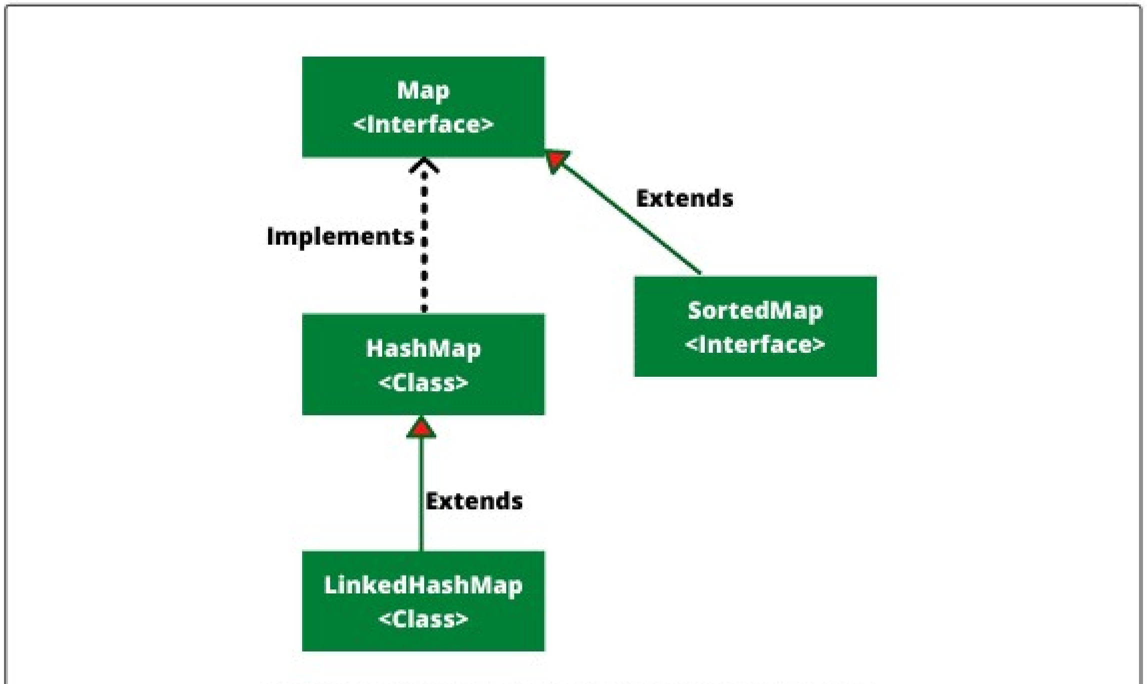


Fig: Hierarchy diagram of LinkedHashMap in Java

### LinkedHashMap class declaration

LinkedHashMap is a generic class that is present in java.util.LinkedHashMap package. It has the following declaration.

```
public class LinkedHashMap<K,V>

  extends HashMap<K,V>

  implements Map<K,V>
```

Here, K defines the type of keys, and V defines the type of values.

### Features of LinkedHashMap in Java

There are several features of LinkedHashMap in Java that should keep in mind. They are as follows:

1. The underlying data structure of LinkedHashMap is HashTable and LinkedList.
2. Java LinkedHashMap maintains the insertion order. The entries in Java LinkedHashMap can be retrieved either in the order in which they were inserted into the map (known as insertion order) or in the order in which they were last accessed, from least to most recently accessed.

3. LinkedHashMap contains unique elements. It contains values based on keys.
4. LinkedHashMap allows only one null key but can have multiple null values.

5. LinkedHashMap in Java is non synchronized. That is, multiple threads can access the same LinkedHashMap object simultaneously.
6. The default initial capacity of LinkedHashMap class is 16 with a load factor of 0.75.

## Constructors of Java LinkedHashMap class

---

Java LinkedHashMap class has the following constructors. They are as follows:

**1. LinkedHashMap():** This constructor is used to create a default LinkedHashMap object. It constructs an empty insertion-ordered LinkedHashMap object with the default initial capacity 16 and load factor 0.75.
The general syntax to construct default LinkedHashMap object is as follows:

```
LinkedHashMap lhmap = new LinkedHashMap();
```

```
or,

LinkedHashMap<K,V> lhmap = new LinkedHashMap<>(); // Generic form
```

**2. LinkedHashMap(int initialCapacity):** It is used to create an empty insertion-ordered LinkedHashMap object with the specified initial capacity and a default load factor of 0.75. The general syntax in generic form is as follows:

```
LinkedHashMap<K,V> lhmap = new LinkedHashMap<>(int initialCapacity);
```

**3. LinkedHashMap(int initialCapacity, float loadFactor):** It is used to create an empty insertion-ordered LinkedHashMap object with the specified initial capacity and load factor. The general syntax in generic form is given below:

```
LinkedHashMap<K,V> lhmap = new LinkedHashMap<>(int initialCapacity, float
loadFactor);

For example:

LinkedHashMap<String, Integer> lhmap = new LinkedHashMap<>(16, 0.75f);
```

**4. LinkedHashMap(Map m):** This constructor is used to create an insertion-ordered LinkedHashMap object with the elements from the given Map m.

**5. LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder):** This constructor is used to create an empty LinkedHashMap instance with the specified initial capacity, load factor, and ordering mode.

If accessOrder is true, access-order is used. If it is false, the insertion order is used. The general syntax to create LinkedHashMap object with three arguments of constructor is as follows:

```
LinkedHashMap<K,V> lhmap = new LinkedHashMap<int initialCapacity, float
loadFactor, boolean accessOrder>();
```

```
For example:

LinkedHashMap<String, String> lhmap = new LinkedHashMap<>(16, 0.75f, true); // For
access order.

LinkedHashMap<String, String> lhmap = new LinkedHashMap<>(16, 0.75f, false); //
For insertion order
```

**Note:** The first four constructors of linked hash map are analogous to the four constructors of hash map class. In each case, the created linked hash map maintains the insertion order.

The last constructor of linked hash map allows us to specify whether elements will be stored in the linked list by insertion order, or by order of last access.

## LinkedHashMap Methods in Java

---

The methods of LinkedHashMap are exactly the same as HashMap class methods, except for one method that is added by LinkedHashMap. This method is removeEldestEntry().

The general syntax for this method is as follows:

```
protected boolean removeEldestEntry(Map.Entry<K,V> e)
```

The parameter e is the least recently added entry in the map, or if it is an access-ordered map, the least recently accessed entry.

This method returns true if the map removes this eldest (oldest) entry. If this entry is retained, or not removed, this method returns false.

The removeEldestEntry() method is called by put() and putAll() after adding a new entry into the map. It helps to remove the eldest entry each time when a new entry is added.

This method is useful if the map represents a cache. It allows the map to reduce memory consumption by deleting stale entries.

## Java LinkedHashMap Example Programs

---

Let's take different example programs where we will perform frequently asked operations based on the methods of LinkedHashMap.

1. Let's create a program where we will perform various operations such as adding elements, checking the size of linked hash map, and linked hash map is empty or not before adding elements into it. Look at the program source code.

**Program source code 1:**

```
import java.util.LinkedHashMap;

public class LinkedHashMapEx1 {

public static void main(String[] args)

{

// Create a LinkedHashMap instance.

 LinkedHashMap<String, Integer> lhmap = new LinkedHashMap<>();



// Checking the size of linked hash map before adding entries.

 int size = lhmap.size();

 System.out.println("Size of LinkedHashMap before adding entries: " +size);
```

```
// Checking linked hash map is empty or not before adding entries.

 boolean isEmpty = lhmap.isEmpty();

 System.out.println("Is LinkedHashMap empty: " +isEmpty);



// Adding entries in linked hash map.

  lhmap.put("John", 30);

  lhmap.put("Peter", 25);

  lhmap.put("Ricky", 23);

  lhmap.put("Deep", 28);

  lhmap.put("Mark", 32);



System.out.println("Display entries in LinkedHashMap");

System.out.println(lhmap);



int size2 = lhmap.size();

System.out.println("Size of LinkedHashMap after adding entries: " +size2);



// Adding null as key and value.

   lhmap.put(null, null);

   System.out.println(lhmap);
 }

Output:

      Size of LinkedHashMap before adding entries: 0

      Is LinkedHashMap empty: true

      Display entries in LinkedHashMap

      {John=30, Peter=25, Ricky=23, Deep=28, Mark=32}

      Size of LinkedHashMap after adding entries: 5

      {John=30, Peter=25, Ricky=23, Deep=28, Mark=32, null=null}
```

2. Let's take a program where we will create LinkedHashMap instance with the third argument of constructor set to true. The mappings En=English, Hi=Hindi, Ta=Tamil, De=German, Fr=French will be inserted into this object and entries will be displayed on the console. Look at the program source code to understand better.

**Program source code 2:**

```
import java.util.LinkedHashMap;

public class LinkedHashMapEx2 {

public static void main(String[] args)

{

 LinkedHashMap<String, String> lhmap = new LinkedHashMap<>(16, 0.75f, true);

  lhmap.put("En", "English");

  lhmap.put("Hi", "Hindi");

  lhmap.put("Ta", "Tamil");
  lhmap.put("De", "German");

  lhmap.put("Fr", "French");



System.out.println("Initially, entries in LinkedHashMap lhmap: " +lhmap);

System.out.println("Value corresponding to key Hi: " +lhmap.get("Hi"));



System.out.println("Value corresponding to key En: " +lhmap.get("En"));

System.out.println("After accessing entries Hi and En: " +lhmap);



System.out.println("\n");



LinkedHashMap<String, String> lhmap2 = new LinkedHashMap<>(16, 0.75f, false);

 lhmap2.put("En", "English");

 lhmap2.put("Hi", "Hindi");

 lhmap2.put("Ta", "Tamil");

 lhmap2.put("De", "German");

 lhmap2.put("Fr", "French");



 System.out.println("Initially, entries in LinkedHashMap lhmap2: " +lhmap2);

 System.out.println("Value corresponding to key Hi: " +lhmap.get("Hi"));
 System.out.println("Value corresponding to key En: " +lhmap.get("En"));

 System.out.println("After accessing entries Hi and En: " +lhmap2);

 }

}

Output:

    Initially, entries in LinkedHashMap lhmap: {En=English, Hi=Hindi, Ta=Tamil,
De=German, Fr=French}

    Value corresponding to key Hi: Hindi

    Value corresponding to key En: English

    After accessing entries Hi and En: {Ta=Tamil, De=German, Fr=French, Hi=Hindi,
En=English}



    Initially, entries in LinkedHashMap lhmap2: {En=English, Hi=Hindi, Ta=Tamil,
De=German, Fr=French}

    Value corresponding to key Hi: Hindi

    Value corresponding to key En: English

    After accessing entries Hi and En: {En=English, Hi=Hindi, Ta=Tamil, De=German,
Fr=French}
```

As you can see in the above program, the mapping Hi=Hindi is accessed, followed by the mapping En=English. After accessing entries, the mappings are displayed in the order that they were last accessed.

Another LinkedHashMap instance is created with the third argument set to false. The same steps are followed here as followed by previous steps. After accessing entries, the insertion order is maintained.

3. Let’s take an example program where we will perform remove, and replace operations. We will also check that a particular value or key is present in linked hash map or not. Look at the following source code.

**Program source code 3:**

```
import java.util.LinkedHashMap;

public class LinkedHashMapEx3 {

public static void main(String[] args)

{

LinkedHashMap<String, String> lhmap = new LinkedHashMap<>();

  lhmap.put("En", "English");

  lhmap.put("Hi", "Hindi");

  lhmap.put("Ta", "Tamil");

  lhmap.put("De", "German");

  lhmap.put("Fr", "French");



System.out.println("Entries in LinkedHashMap lhmap: " +lhmap);



// Call remove() method to delete an entry for specified key.

   lhmap.remove("De");
   System.out.println("Updated Entries in LinkedHashMap: " +lhmap);



// Call replace() method to replace specified value for a specified key.

   lhmap.replace("En", "English-US");

   System.out.println("After replacing, updated entries in LinkedHashMap: " +lhmap);



// Call containsValue() method to determine specified value for specified key.

   boolean value = lhmap.containsValue("Hindi");

   System.out.println("Is Hindi present in LinkedHashMap: " +value);

 }

}

Output:

     Entries in LinkedHashMap lhmap: {En=English, Hi=Hindi, Ta=Tamil, De=German,
Fr=French}

     Updated Entries in LinkedHashMap: {En=English, Hi=Hindi, Ta=Tamil, Fr=French}

     After replacing, updated entries in LinkedHashMap: {Hi=Hindi, Ta=Tamil,
Fr=French, En=English-US}

     Is Hindi present in LinkedHashMap: true
```

## How to iterate LinkedHashMap in Java?

# Java Map Interface Implementation Classes

Let’s take an example program where we will iterate entries, keys, and values of LinkedHashMap using java iterator concept. Look at the source code.

**Program source code 4:**

```
import java.util.Iterator;

import java.util.LinkedHashMap;

import java.util.Map.Entry;

public class IteratingLinkedHashMap {



public static void main(String[] args)

{

 LinkedHashMap<Character, String> lhmap = new LinkedHashMap<>();

  lhmap.put('R', "Red");

  lhmap.put('G', "Green");

  lhmap.put('B', "Brown");

  lhmap.put('O', "Orange");

  lhmap.put('P', "Pink");



System.out.println("Entries in LinkedHashMap lhmap: " +lhmap);



Iterator<Entry<Character, String>> itr = lhmap.entrySet().iterator(); // entrySet is a
method that is used to get view of entries of a linked hash map.
```

```
System.out.println("Iterating Entries of LinkedHashMap");

while(itr.hasNext())

{

 Object key = itr.next();

 System.out.println(key);

}

System.out.println("\n");



Iterator<Character> itr2 = lhmap.keySet().iterator(); // keySet is a method that is used to
get view of keys of a linked hash map.

System.out.println("Iterating Keys of LinkedHashMap");

while(itr2.hasNext())

{

 Object keyView = itr2.next();

 System.out.println(keyView);

}

System.out.println("\n");



Iterator<String> itr3 = lhmap.values().iterator(); // values is a method that is used to get
values of keys of a linked hash map.

System.out.println("Iterating Values of LinkedHashMap");
```

```
while(itr3.hasNext())

{

 Object valuesView = itr3.next();

 System.out.println(valuesView);

  }

 }

}

Output:

      Entries in LinkedHashMap lhmap: {R=Red, G=Green, B=Brown, O=Orange,
P=Pink}

      Iterating Entries of LinkedHashMap

      R=Red

      G=Green

      B=Brown

      O=Orange

      P=Pink



      Iterating Keys of LinkedHashMap

      R

      G
```

```
B

O

P



Iterating Values of LinkedHashMap

Red

Green

Brown

Orange

Pink
```

Let’s iterate entries of linked hashmap using forEach() method.

**Program source code 5:**

```
import java.util.LinkedHashMap;

public class IteratingLinkedHashMap2 {

public static void main(String[] args)

{

LinkedHashMap<Character, String> lhmap = new LinkedHashMap<>();

  lhmap.put('R', "Red");

  lhmap.put('G', "Green");

  lhmap.put('B', "Brown");
  lhmap.put('O', "Orange");

  lhmap.put('P', "Pink");


System.out.println("Iterating Entries of LinkedHashMap");


// Iteration over map using forEach() method.

  lhmap.forEach((k,v) -> System.out.println("Color code: "+ k + ", Color name: " + v));

 }

}

Output:

      Iterating Entries of LinkedHashMap

      Color code: R, Color name: Red

      Color code: G, Color name: Green

      Color code: B, Color name: Brown

      Color code: O, Color name: Orange

      Color code: P, Color name: Pink
```

## Program based on removeEldestEntry() method

---

Let's take an example program based on removeEldestEntry() method.

**Program source code 6:**

## Java Map Interface Implementation Classes

```
import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapEx1 {

public static void main(String[] args)

{

final int max = 5;

LinkedHashMap<String, String> lhmap = new LinkedHashMap<String,String>(){

  protected boolean removeEldestEntry(Map.Entry<String, String> e)

  {

    return size() > max;

  }

};

  lhmap.put("R", "Red");

  lhmap.put("G", "Green");

  lhmap.put("B", "Brown");

  lhmap.put("O", "Orange");

  lhmap.put("P", "Pink");


System.out.println("Initial Entries of LinkedHashMap");

System.out.println(lhmap);
// Adding more entry into linked hash map.

  lhmap.put("W", "White");

  System.out.println("Displaying Map after adding more entry: " +lhmap);



 lhmap.put("Y", "Yellow"); // Adding one more entry into linked hash map.

 System.out.println("Displaying Map after adding one more entry: " +lhmap);

 }

}

Output:

     Initial Entries of LinkedHashMap

     {R=Red, G=Green, B=Brown, O=Orange, P=Pink}

     Displaying Map after adding more entry: {G=Green, B=Brown, O=Orange, P=Pink,
W=White}

     Displaying Map after adding one more entry: {B=Brown, O=Orange, P=Pink,
W=White, Y=Yellow}
```

## When to use LinkedHashMap in Java?

LinkedHashMap can be used when you want to preserve the insertion order. Java LinkedHashMap is slower than HashMap but it is suitable when more number of insertions and deletions happen.

# Java Map Interface Implementation Classes

## Which implementation is better to use: HashMap or LinkedHashMap?

Both HashMap and LinkedHashMap classes provide comparable performance but HashMap is a natural choice if the ordering of elements is not an issue.

Adding, removing, and finding entries in a LinkedHashMap is slightly slower than in HashMap because it needs to maintain the order of doubly linked list in addition to the hashed data structure.

## TreeMap in Java | Methods, Example

**TreeMap in Java** is a concrete class that is a red-black tree based implementation of the Map interface.
It provides an efficient way of storing key/value pairs in sorted order automatically and allows rapid retrieval. It was added in JDK 1.2 and present in java.util.TreeMap.

A TreeMap implementation provides guaranteed log(n) time performance for checking, adding, retrieval, and removal operations.

The two main difference between HashMap and TreeMap is that

- HashMap is an unordered collection of elements while TreeMap is sorted in the ascending order of its keys. The keys are sorted either using Comparable interface or Comparator interface.
- HashMap allows only one null key while TreeMap does not allow any null key.

### Hierarchy of TreeMap in Java

## Java Map Interface Implementation Classes

TreeMap class is the subclass of AbstractMap and implements Map, SortedMap (a subinterface of Map interface), and NavigableMap interfaces.
TreeMap also implements Cloneable and Serializable interfaces. The hierarchy diagram of TreeMap in Java is shown in the below figure.

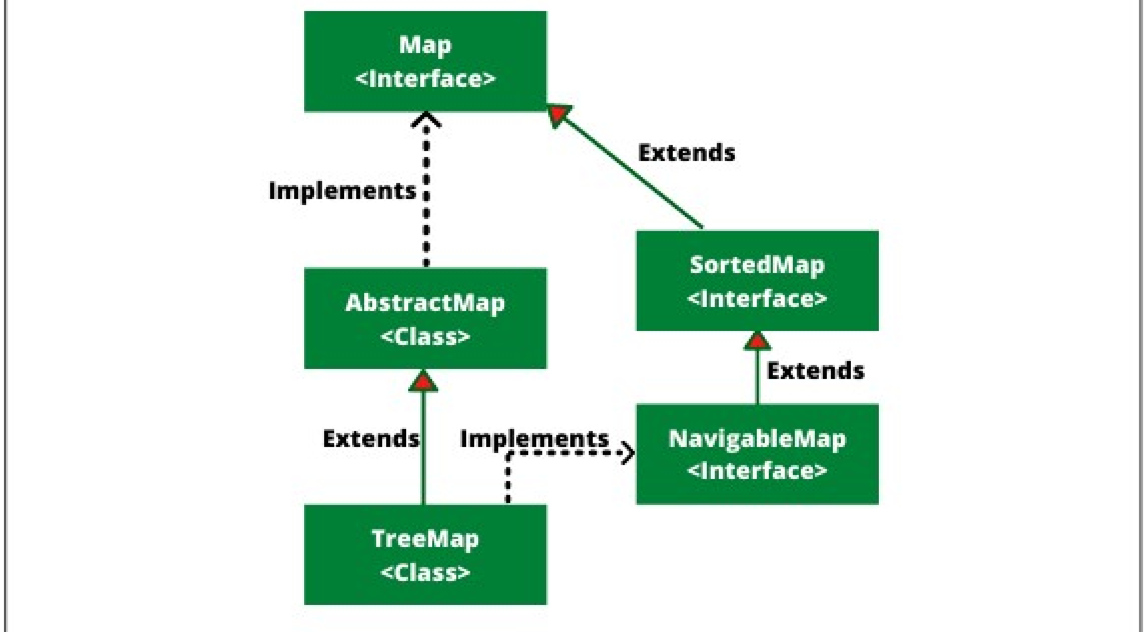


Fig: Hierarchy diagram of TreeMap in Java

### TreeMap class declaration

---

TreeMap is a generic class that can be declared as follows:

```
public class TreeMap<K,V>

  extends AbstractMap<K,V>

  implements NavigableMap<K,V>, Cloneable, Serializable
```

Here, K defines the type of keys, and V defines the type of values.

## Important Features of  TreeMap

There are several important features of TreeMap in Java that should be kept in mind. They are as follows:

1. The underlying data structure of Java TreeMap is a red-black binary search tree.

2. TreeMap contains only unique elements.

3. TreeMap cannot have a null key but can have multiple null values.

4. Java TreeMap is non synchronized. That means it is not thread-safe. We can create a synchronized version of map by calling Collections.synchronizedMap() on the map.

5. TreeMap in Java maintains ascending order. The mappings are sorted in treemap either according to the natural ordering of keys or by a comparator that is provided during the object creation of TreeMap depending upon the constructor used.

6. Java TreeMap determines the order of entries by using either Comparable interface or Comparator class.

7. The iterator returned by TreeMap is fail-fast. That means we cannot modify map during iteration.

## Constructors of TreeMap class in Java

Java TreeMap defines the following constructors. They are as follows:

# Java Map Interface Implementation Classes

**1. TreeMap():** This constructor is used to create an empty TreeMap that will be sorted according to the natural order of its key. All keys added to tree map must implement Comparable interface.
The compareTo() method in the Comparable interface is used to compare keys in the map. If you put a string key into the map whose type is an integer, the put() method will throw an exception named ClassCastException.

**2. TreeMap(Comparator c):** This form of constructor is used to create an empty tree-based map. All keys inserted in the map will be sorted according to the given Comparator c.
The compare() method in the comparator is used to order the entries in the map based on keys.

**3. TreeMap(Map m):** This form of constructor is used to initialize a tree map with entries from Map m that will be sorted according to the natural order of the keys.
**4. TreeMap(SortedMap sm):** This constructor is used to initialize a treemap with the entries from the SortedMap sm which will be sorted according to the same ordering as sm.

## Methods of Java TreeMap class

---

TreeMap class in java provides a variety of methods to perform different tasks. They are as follows:

1. **void clear():** This method removes all objects (entries) from the tree map.
2. **V put(K key, V value):** This method is used to insert a key/value pair in the tree map.
3. **void putAll(Map m):** It is used to add key/value pairs from Map m to the current tree map.

4. **V remove(Object key):** It is used to remove the key-value pair of the specified key from the tree map.
5. **V get(Object key):** This method is used to retrieve the value associated with key. If key is null, it will throw NullPointerException.
6. **K firstKey():** It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.

7. **K lastKey():** It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.

8. **boolean containsKey(Object key):** This method returns true if the tree map contains a particular key.
9. **boolean containsValue(Object value):** This method returns true if the tree map contains a particular value.
10. **int size():** This method returns the number of entries (objects) in the tree map.
11. **Set keySet():** This method returns a set (collection) of all keys of the tree map.
12. **Set entrySet():** This method returns a set view containing all key/value pairs in the tree map.
13. **Collection values():** It returns a collection view of all values contained in the tree map.
14. **Comparator comparator():** It is used to retrieve map’s comparator that arranges the key in order, or null if the map uses the natural ordering of elements.
15. **Object clone():** It is used to retrieve the copy of the tree map without cloning its elements.
16. **Map.Entry<K, V> ceilingEntry(K key):** This method returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.

17. **K ceilingKey(K key):** This method returns the least key, greater than or equal to the specified key, or null if there is no such key.
Both ceilingEntry() and ceilingKey() methods throw ClassCastException if key cannot be compared with the current key in the map. They will throw NullPointerException when key is null because TreeMap does not permit null key.

18. **NavigableSet<K> descendingKeySet():** It returns a reverse order navigable set-based view of the keys contained in the map. The iterator of set returns the keys in descending order.
19. **NavigableMap<K,V> descendingMap():** It returns a reverse order view of this map.
20. **Map.Entry firstEntry():** It returns the key-value pair having the least key in the map or null if the map is empty.
21. **Map.Entry<K,V> floorEntry(K key):** It returns a key-value pair associated with the greatest key less than or equal to the specified key, or null when there is no such key.
22. **K floorKey(K Key):** It returns the greatest key less than or equal to the specified key, or null when there is no such key.
23. **SortedMap<K,V> headMap(K toKey):** This method returns the key-value pairs (a view) of that portion of this map whose keys are strictly less than toKey.
24. **NavigableMap<K,V> headMap(K toKey, boolean inclusive):** This method returns key-value pairs (view) of that portion of this map whose keys are less than (or equal to if inclusive is true) toKey.
Both headMap() methods will throw ClassCastException if toKey is not compatible with this map's comparator. If to Key is null, this method throws NullPointerException because TreeMap does not allow null key.

25. **Map.Entry<K,V> higherEntry(K key):** This method returns a key-value pair or mapping corresponding to the least key strictly greater than the given key, or null if there is no such key.
26. **K higherKey(K key):** It is used to retrieve the least key that is strictly greater than the specified key, or null when there is no such key.

Both these methods throw ClassCastException when key cannot be compared with the current key in the map and NullPointerException if key is null.

27. **Map.Entry<K,V> lastEntry():** It returns the key-value pair corresponding to the greatest key in the map, or null if the map is empty or there is no such key.
28. **Map.Entry<K,V> lowerEntry(K key):** This method returns a key-value pair associated with the greatest key strictly less than the specified key, or null if there is no such key.
29. **K lowerKey(K key):** It returns the greatest key strictly less than the specified key, or null if there is no such key.
Both lowerEntry() and lowerKey() methods throw ClassCastException if key cannot be compared with the keys currently in the map, and NullPointerException when key is null.

30. **NavigableSet<K> navigableKeySet():** It returns a navigable set view of the keys contained in this map.
31. **Map.Entry<K,V> pollFirstEntry():** It removes and returns a key-value pair corresponding to the least key in this map, or null if the map is empty.
32. **Map.Entry<K,V> pollLastEntry():** It removes and returns a key-value pair corresponding to the greatest key in this map, or null if the map is empty.
33. **V replace(K key, V value):** It is used to replace the specified value for a specified key.
34. **boolean replace(K key, V oldValue, V newValue):** It is used to replace the old value with the new value for a specified key.
35. **NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive):** It returns key-value mapping from the map whose keys range from fromKey to toKey.
36. **SortedMap<K,V> subMap(K fromKey, K toKey):** It returns key-value mapping or pairs whose keys range from fromKey, inclusive, to toKey, exclusive.

37. **SortedMap<K,V> tailMap(K fromKey):** This method returns key-value pairs whose keys are greater than or equal to fromKey.
38. **NavigableMap<K,V> tailMap(K fromKey, boolean inclusive):** This method returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey.

## Java TreeMap Example Programs

---

Let's take various example programs where we will perform some useful operations based on the methods of TreeMap.

1. Let's take a program where we will create HashMap, LinkedHashMap, and TreeMap objects and see the order of entries of the map.

**Program source code 1:**

```
import java.util.HashMap;

import java.util.LinkedHashMap;

import java.util.TreeMap;

public class TreeMapEx1 {

public static void main(String[] args)

{

// Create a HashMap.

  HashMap<String, String> hmap = new HashMap<>();



   hmap.put("R", "Red");

   hmap.put("G", "Green");
   hmap.put("B", "Brown");

   hmap.put("O", "Orange");

   hmap.put("P", "Pink");



System.out.println("Entries of HashMap: " +hmap);



// Create a TreeMap from the previous HashMap.

 TreeMap<String, String> tmap = new TreeMap<>(hmap);



 System.out.println("Entries in ascending order of keys: " +tmap);



// Create a LinkedHashMap.

 LinkedHashMap<String, String> lhmap = new LinkedHashMap<>();



  lhmap.put("R", "Red");

  lhmap.put("G", "Green");

  lhmap.put("B", "Brown");

  lhmap.put("O", "Orange");

  lhmap.put("P", "Pink");
```

```
System.out.println("Entries in LinkedHashMap: " +lhmap);

 }

}

Output:

      Entries of HashMap: {P=Pink, R=Red, B=Brown, G=Green, O=Orange}

      Entries in ascending order of keys: {B=Brown, G=Green, O=Orange, P=Pink,
R=Red}

      Entries in LinkedHashMap: {R=Red, G=Green, B=Brown, O=Orange, P=Pink}
```

As you can see in the output of the program, HashMap does not maintain the order, LinkedHashMap maintains the order of entries as we inserted, while TreeMap sorted entries in ascending order of its keys.
2. Let’s create a program and perform add, remove, and replace operations in the map.

**Program source code 2:**

```
import java.util.TreeMap;

public class TreeMapEx2{

public static void main(String[] args)

{

TreeMap<String, Integer> tmap = new TreeMap<>();

int size = tmap.size();

System.out.println("Size of TreeMap before adding entries: " +size);



boolean isEmpty = tmap.isEmpty();
System.out.println("Is TreeMap empty: " +isEmpty);



// Adding entries in tree map.

  tmap.put("John", 25);

  tmap.put("Ricky", 22);

  tmap.put("Deep", 28);

  tmap.put("Mark", 20);

  tmap.put("Peter", 30);



System.out.println("Entries in ascending order: " +tmap);

 tmap.remove("Mark");

System.out.println("Entries of TreeMap after removing: " +tmap);



 tmap.replace("Peter", 18);

 System.out.println("Updated enrties of TreeMap: " +tmap);

 }

}

Output:

      Size of TreeMap before adding entries: 0

      Is TreeMap empty: true
```

```
    Entries in ascending order: {Deep=28, John=25, Mark=20, Peter=30, Ricky=22}

    Entries of TreeMap after removing: {Deep=28, John=25, Peter=30, Ricky=22}

    Updated enrties of TreeMap: {Deep=28, John=25, Peter=18, Ricky=22}
```

3. Let's take an example program based on entrySet(), keySet(), values(), get(), containsKey(), and containsValue() methods.

**Program source code 3:**

```
import java.util.TreeMap;

public class TreeMapEx1{

public static void main(String[] args)

{

TreeMap<Character, String> tmap = new TreeMap<>();



  tmap.put('A', "Apple");

  tmap.put('P', "Parot");

  tmap.put('C', "Cat");

  tmap.put('B', "Boy");

  tmap.put('D', "Dog");



Object entrySet = tmap.entrySet();

System.out.println("Entry set: " +entrySet);

System.out.println("Key set: " +tmap.keySet());
System.out.println("Value set: " +tmap.values());



Object vGet = tmap.get('C');

System.out.println("C: " +vGet);



boolean containsKey = tmap.containsKey('B');

System.out.println("Is key 'B' present in map: " +containsKey);



boolean containsValue = tmap.containsValue("Apple");

System.out.println("Is Apple present in map: " +containsValue);

 }

}

Output:

    Entry set: [A=Apple, B=Boy, C=Cat, D=Dog, P=Parot]

    Key set: [A, B, C, D, P]

    Value set: [Apple, Boy, Cat, Dog, Parot]

    C: Cat

    Is key 'B' present in map: true

    Is Apple present in map: true
```

## Java Map Interface Implementation Classes

4. Let’s take an example program based on ceilingEntry(), ceilingKey(), firstEntry(), lastEntry(), floorEntry() methods.

**Program source code 4:**

```
import java.util.TreeMap;

public class TreeMapEx1{

public static void main(String[] args)

{

TreeMap<Integer, String> tmap = new TreeMap<>();

  tmap.put(25, "John");

  tmap.put(22, "Shubh");

  tmap.put(30, "Ricky");

  tmap.put(35, "Peter");

  tmap.put(18, "Johnson");



System.out.println("ceilingEntry: " +tmap.ceilingEntry(32));

System.out.println("ceilingKey: " +tmap.ceilingKey(32));



System.out.println("firstEntry: " +tmap.firstEntry());

System.out.println("lastEntry: " +tmap.lastEntry());



System.out.println("floorEntry: " +tmap.floorEntry(31));
System.out.println("HigherEntry: " +tmap.higherEntry(30));

System.out.println("LowerEntry: " +tmap.lowerEntry(30));



System.out.println("pollFirstEntry: " +tmap.pollFirstEntry());

System.out.println("pollLastEntry: " +tmap.pollLastEntry());

 }

}

Output:

    ceilingEntry: 35=Peter

    ceilingKey: 35

    firstEntry: 18=Johnson

    lastEntry: 35=Peter

    floorEntry: 30=Ricky

    HigherEntry: 35=Peter

    LowerEntry: 25=John

    pollFirstEntry: 18=Johnson

    pollLastEntry: 35=Peter
```

5. Let’s take one more example program based on headMap(), tailMap(), subMap() methods of SortedMap, and NavigableMap interfaces.

**Program source code 5:**

```
import java.util.TreeMap;
```

## Java Map Interface Implementation Classes

```
public class TreeMapEx1{

public static void main(String[] args)

{

TreeMap<Integer, String> tmap = new TreeMap<>();



  tmap.put(25, "John");

  tmap.put(22, "Shubh");

  tmap.put(30, "Ricky");

  tmap.put(35, "Peter");

  tmap.put(18, "Johnson");



System.out.println("Sorted tree map: " +tmap);



// Use methods of NavigableMap interface.

System.out.println("Descending order of tree map: " +tmap.descendingMap());

System.out.println("headMap: "+tmap.headMap(22,true));  // Returns entries whose keys
are less than or equal to the specified key.



System.out.println("tailMap: "+tmap.tailMap(22,true)); // Returns entries whose keys are
greater than or equal to the specified key.
System.out.println("subMap: "+tmap.subMap(18, false, 35, true)); // Returns entries exists in between the specified key.

System.out.println("\n");

// Use methods of SortedMap interface.

System.out.println("headMap: "+tmap.headMap(40)); // Returns entries whose keys are less than the specified key.

System.out.println("tailMap: "+tmap.tailMap(22)); // Returns entries whose keys are greater than or equal to the specified key.

System.out.println("subMap: "+tmap.subMap(19, 25)); // Returns entries exists in between the specified key.

 }

}
```

Output:

```
    Sorted tree map: {18=Johnson, 22=Shubh, 25=John, 30=Ricky, 35=Peter}

    Descending order of tree map: {35=Peter, 30=Ricky, 25=John, 22=Shubh, 18=Johnson}

    headMap: {18=Johnson, 22=Shubh}

    tailMap: {22=Shubh, 25=John, 30=Ricky, 35=Peter}

    subMap: {22=Shubh, 25=John, 30=Ricky, 35=Peter}
```

```
headMap: {18=Johnson, 22=Shubh, 25=John, 30=Ricky, 35=Peter}

tailMap: {22=Shubh, 25=John, 30=Ricky, 35=Peter}

subMap: {22=Shubh}
```

## When to use TreeMap in Java?

TreeMap is slower than HashMap but it is preferred:

- When we do not want null key in the map.
- When we want the order of entries in sorted ascending order.

**Key points:**

1. HashTable is suitable when you are not working in a multithreading environment.
2. HashMap is slightly better than HashTable but it is not thread-safe. It is suitable if the order of elements is not an issue.
3. TreeMap is slower than HashMap but it is suitable when you need the map in sorted ascending order.
4. LinkedHashMap is also slower than HashMap, but it is preferred if more number of insertions and deletions happen on the map.

## WeakHashMap in Java | Methods, Example

**WeakHashMap in Java** is a class that provides another Map implementation based on weak keys. It was added in JDK 1.2 version and present in java.util.WeakHashMap.

# Java Map Interface Implementation Classes

Java WeakHashMap extends AbstractMap class to use HashTable with weak keys. The key of WeakHashMap has a weak reference that allows an entry in a map to be garbage-collected when its key is no longer in use.

In simple words, an entry in a WeakHashMap will be automatically removed from the map by the garbage-collector when its key is unused.

Garbage-collector clears all weak references to the keys(inside and outside of the map) when keys are no longer in ordinary use.

## Hierarchy of WeakHashMap class in Java

---

WeakHashMap class extends AbstractMap class that implements Map interface. The hierarchy diagram of WeakHashMap class is shown in the below figure.

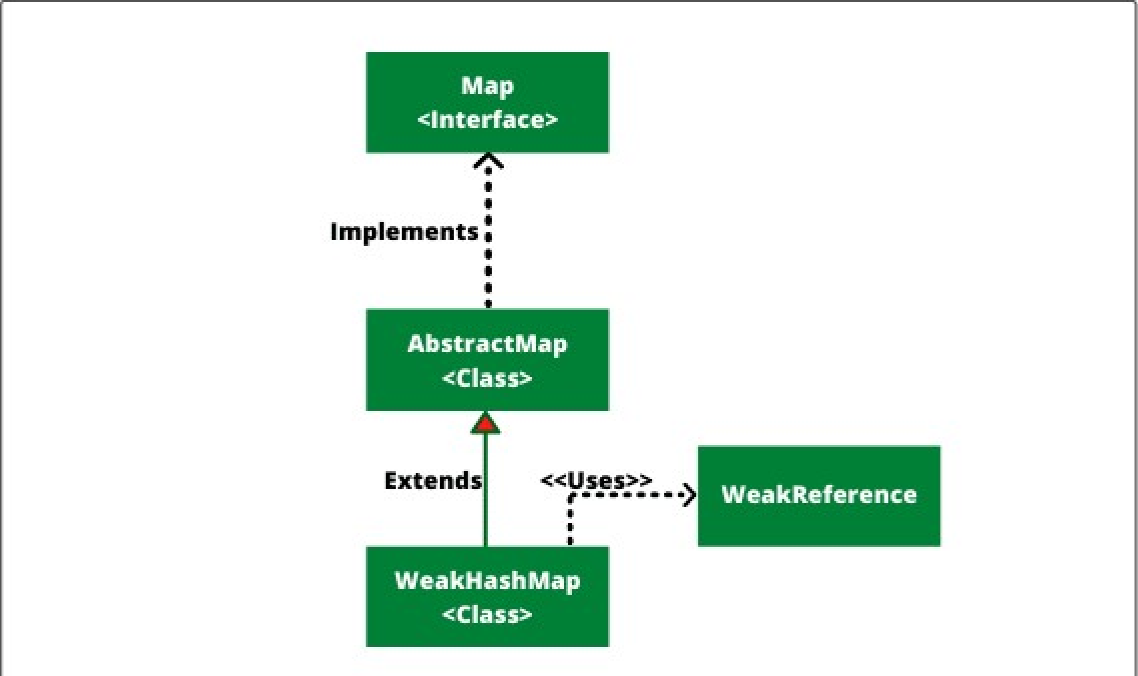


Fig: Hierarchy diagram of WeakHashMap in Java

## WeakHashMap class declaration

WeakHashMap is a generic class that is declared as follows:

```
public class WeakHashMap<K,V>

  extends AbstractMap<K,V>

    implements Map<K,V>
```

Here, K defines the type of Keys and V defines the type of Values.

## Features of WeakHashMap

There are several important features of WeakHashMap in Java that should keep in mind. They are as follows:

1. Garbage collector may remove keys and their associated values from the WeakHashMap at any time in java.
2. It enables us to store data in form of key-value pairs in java.
3. Java WeakHashMap does not allow storing duplicate keys.
4. It allows to insert only one null key in the map but several null values can be added.
5. WeakHashMap does not maintain insertion order in java.

6. WeakHashMap in Java is not synchronized. So, it is not thread-safe. Two or more threads on the same WeakHashMap object can access at the same time in java.

We can synchronize weakHashMap by using Collections' class synchronizedList() method.

```
Map synchronizedMap = Collections.synchronizedMap(weakHashMap);
```

Now, no two threads can access the same weakhashmap object simultaneously.

## Internal working of WeakHashMap in Java

In this section, we will learn how WeakHashMap works internally in java. WeakHashMap uses a weak reference to hold keys. An object can have both weak and strong (normal) references. If an object has a strong reference, it can never be garbage collected.

A weak reference is a reference that by itself does not prevent an object from being garbage collected. If an object has only weak references or no references then it is collected by garbage collector.

WeakHashMap periodically checks objects that have weak references. A weak reference signifies that the key is no longer used by anyone and is collected by garbage collector. Then WeakHashMap removes the associated entry from the map.

## Constructors of WeakHashMap class

WeakHashMap class defines four constructors that are as:

**1. WeakHashMap():** This form of constructor creates a new, empty WeakHashMap with the default initial capacity (16) and load factor (0.75).
**2. WeakHashMap(int initialCapacity):** This form of constructor creates a new, empty WeakHashMap with the specified initial capacity and default load factor (0.75).

**3. WeakHashMap(int initialCapacity, float loadFactor):** This constructor is used to create a new, empty WeakHashMap with the given initial capacity and the given load factor.
**4. WeakHashMap(Map m):** It constructs a new WeakHashMap with the same mappings as the specified map.

## WeakHashMap Methods in Java

---

The methods of WeakHashMap is exactly the same as HashMap's methods. To know more detail about HashMap methods, click on the tutorial: HashMap in Java | Methods, Example

1. Methods inherited from class java.util.AbstractMap
clone(), equals(), hashCode(), toString()

2. Methods inherited from class java.lang.Object
finalize(), getClass(), notify(), notifyAll(), wait()

3. Methods inherited from interface java.util.Map
compute(), computeIfAbsent(), computeIfPresent(), equals(), hashCode(), merge(), putIfAbsent(), remove(), replace()

## When to use WeakHashMap in Java?

---

WeakHashMap can be used when:

1. We want to remove keys and their associated values automatically by garbage-collector when keys are no longer in use.

2. We want to store data in key-value pair form on the map.

3. We want to store only one null key.

4. We don't care about insertion order in java.

5. It can be used when we are not working in a multithreading environment in java.

## IdentityHashMap in Java | Methods, Example

**IdentityHashMap in Java** is a concrete class that is similar to HashMap except that it uses reference equality (==) when comparing entries (keys and values).
In simple words, Java IdentityHashMap uses reference equality (==) instead of object equality (equals() method) when comparing keys and values.

While comparing keys, HashMap uses the equals() method whereas IdentityHashMap uses reference equality (==).

In HashMap, two keys k1 and k2 are considered equal if and only if (k1==null?k2==null:k1.equals(k2)). In IdentityHashMap, two keys are considered equal if and only if (k1==k2).

IdentityHashMap in Java was added in JDK 1.4 version and is present in java.util.IdentityHashMap package. This class is rarely used in the program.

### Hierarchy of IdentityHashMap in Java

---

Java IdentityHashMap class extends AbstractMap class and implements Map interface (by using a hash table). The hierarchy diagram of IdentityHashMap is shown in the below figure.

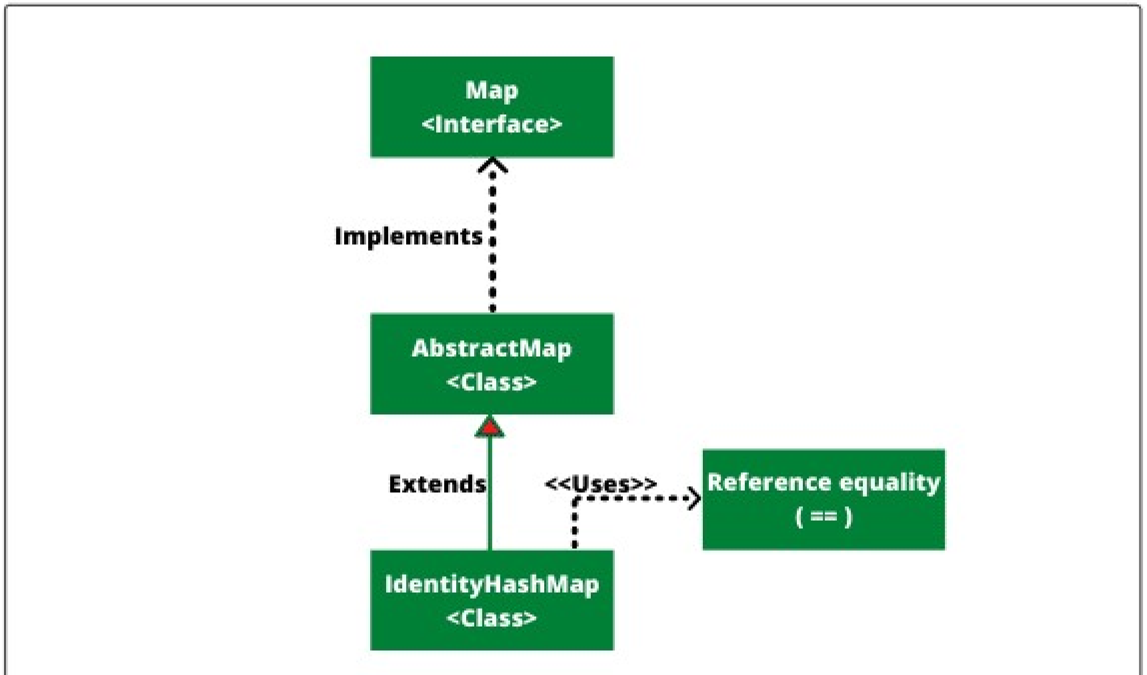


Fig: Hierarchy diagram of IdentityHashMap in Java

## IdentityHashMap class declaration

IdentityHashMap is a generic class that can be declared as follows:

```
class IdentityHashMap<K,V>

 extends AbstractMap<K,V>

  implements Map<K,V>
```

Here, K defined the type of keys and V defines the type of Values. The API documentation explicitly states that IdentityHashMap is not for general use.

## Features of IdentityHashMap

# Java Map Interface Implementation Classes

There are several important features of IdentityHashMap in Java that should keep in mind. They are as:

1. IdentityHashMap allows us to store data in the form of key-value pairs.

2. In IdentityHashMap, only one null key is allowed to add but many null values can be inserted.

3. Like HashMap, IdentityHashMap does not maintain insertion order. There is no guarantee that elements will be retrieved in the same order as we have inserted.

4. It is not synchronized. That means it is not thread-safe. So, two or more threads on same IdentityHashMap object can access it at the same time.
We can synchronize IdentityHashMap by using Collections's class synchronizedMap() method. The syntax to synchronize it is as follows:

```
Map synchronizedMap = Collections.synchronizedMap(IdentityHashMap);
```

Now, no two or more threads can access the same object of map simultaneously.

5. Iterator returned by IdentityHashMap is a fail-fast. That means any structural modification made to IdentityHashMap like adding or removing elements during Iteration will throw an exception named **ConcurrentModificationException**.

## Constructors of IdentityHashMap class

Java IdentityHashMap has the following three public constructors:

- **public IdentityHashMap()**

- **public IdentityHashMap(int ems)**
- **public IdentityHashMap(Map m)**

The first two constructors are standard that are used for general-purpose Map implementation. They create an empty IdentityHashMap with expected maximum size of 21 (by default) and ems. The expected maximum size ems is the maximum number of entries that are expected to hold.

The third constructor creates IdentityHashMap containing the same entries that are present in Map m.

## IdentityHashMap Methods in Java

IdentityHashMap methods are exactly the same as HashMap's methods. So, let's perform some operations based on the IdentityHashMap methods and see the difference between HashMap and IdentityHashMap.

## Java IdentityHashMap Example Programs

Let's create a program where we will see the difference between HashMap and IdentityHashMap. Look at the following source code to understand better.

**Program source code 1:**

```
import java.util.HashMap;

import java.util.IdentityHashMap;



public class IdentityHashMapEx1  {
```

```
public static void main(String[] args)

{

// Create a HashMap.

 HashMap<String, String> hmap = new HashMap<>();



 String s1 = new String("R");

 String s2 = new String("R");



// Adding entries in HashMap.

  hmap.put(s1, "Red");

  hmap.put(s2, "Red");



System.out.println("Keys present in HashMap hmap: " +hmap.keySet());

System.out.println("Values present in HashMap hmap: " +hmap.values());



// Create a IdentityHashMap object.

 IdentityHashMap<String, String> ihmap = new IdentityHashMap<>();

  ihmap.put(s1, "Red");

  ihmap.put(s2, "Red");
System.out.println("\n");

System.out.println("Keys present in IdentityHashMap: " +ihmap.keySet());

System.out.println("Values present in IdentityHashMap: " +ihmap.values());

 }

}

Output:

    Keys present in HashMap hmap: [R]

    Values present in HashMap hmap: [Red]



    Keys present in IdentityHashMap: [R, R]

    Values present in IdentityHashMap: [Red, Red]
```

**Explanation:**

1. In the preceding program, an empty HashMap is created. Then two separate strings s1 and s2, each pointing to R, are created. They are used as keys having corresponding values R and put both keys s1 and s2 in the hash map along with their values.

2. Then keys are retrieved using keySet() method and displayed on the console. In the output, you will observe that only one key R is displayed on the console.

3. Values are retrieved using values() method and displayed on the console. The value Red is displayed on the screen.

This is because HashMap uses equals() method for comparing keys. Since both s1 and s2 contain the same contents (i.e. Red), therefore, equals() method will return true.

When an attempt is made to put s2 along with its associated value, it replaces the value. Thus, only one entry (key-value pair) is retained.

4. Now an empty IdentityHashMap ihmap is created. Both s1 and s2 are inserted in the identity hash map object along with their values.

5. Then keySet() method is used to retrieve keys. Both keys, each having values R, are displayed on the screen.

6. values() method is used to retrieve values. Both values, each containing value "Red", are displayed on the screen. As you can see in the output.

This is because IdentityHashMap uses == operator for comparing keys. Since both s1 and s2 are two different objects (although they contain the same value "Red"), the == operator will return false.

An attempt is made to put s2 along with its value in the IdentityHashMap, the second key-value pair will be considered as different key-value pair and added to the identity hash map.

Although, Java IdentityHashMap implements Map interface, still, this class violates Map's general contract of using equals() method during the comparison of key objects.

Therefore, it is not for general use. IdentityHashMap can be used only in cases where reference-equality semantic are required.

2. Let's take an example program based on the following methods of IdentityHashMap. We will perform the following operations. Look at the source code.

**Program source code 2:**

```
import java.util.IdentityHashMap;

public class IdentityHashMapEx2 {

public static void main(String[] args)

{
 String s1 = new String("R");

 String s2 = new String("G");



// Create a IdentityHashMap object.

IdentityHashMap<String, String> ihmap = new IdentityHashMap<>();



  ihmap.put(s1, "Red");

  ihmap.put(s2, "Green");



System.out.println("Keys present in IdentityHashMap: " +ihmap.keySet());

System.out.println("Values present in IdentityHashMap: " +ihmap.values());



// Inserting some more keys and values.

  ihmap.put("P", "Pink");

  ihmap.put("B", "Black");

  ihmap.put("W", "White");

  ihmap.put("O", "Orange");



System.out.println("\n");

System.out.println("IdentityHashMap contains key P: " +ihmap.containsKey("P"));
System.out.println("IdentityHashMap contains value Orange: "
+ihmap.containsValue("Orange"));

System.out.println("Set of mappings contained in ihmap: " +ihmap.entrySet());



System.out.println("Value corresponding to key W: " +ihmap.get("W"));

System.out.println("Is IdentityHashMap ihmap empty: " +ihmap.isEmpty());



ihmap.clear();

System.out.println("After using clear() method ihmap contains: " +ihmap);

  }

}
```

Output:

```
    Keys present in IdentityHashMap: [R, G]

    Values present in IdentityHashMap: [Red, Green]



    IdentityHashMap contains key P: true

    IdentityHashMap contains value Orange: true



    Set of mappings contained in ihmap: [R=Red, P=Pink, O=Orange, W=White,
B=Black, G=Green]
```

```
Value corresponding to key W: White

Is IdentityHashMap ihmap empty: false

After using clear() method ihmap contains: {}
```

### When to use IdentityHashMap in Java?

IdentityHashMap can be used when:

- We want to perform deep-copying.
- We want to maintain proxy objects.
- We want to store only one null key on the map.
- We do not want the insertion order of entries.
- We are not working in a multithreading environment.

## EnumMap in Java | Methods, Example

**EnumMap in Java** is a concrete class that provides Map implementation whose keys are of the enum type.
All the keys in the EnumMap must originate from a single enum type that is specified, implicitly or explicitly, when the map is created.

Enum maps are represented internally as an array to hold the corresponding values that are extremely compact and efficient.

EnumMap in Java was added in Java 5.0 version and present in java.util.EnumMap package.

### Hierarchy of EnumMap class in Java

Java EnumMap class extends AbstractMap class and implements Map, Serializable and Cloneable interfaces. The hierarchy diagram of

# Java Map Interface Implementation Classes

EnumMap in Java is shown in the below diagram.

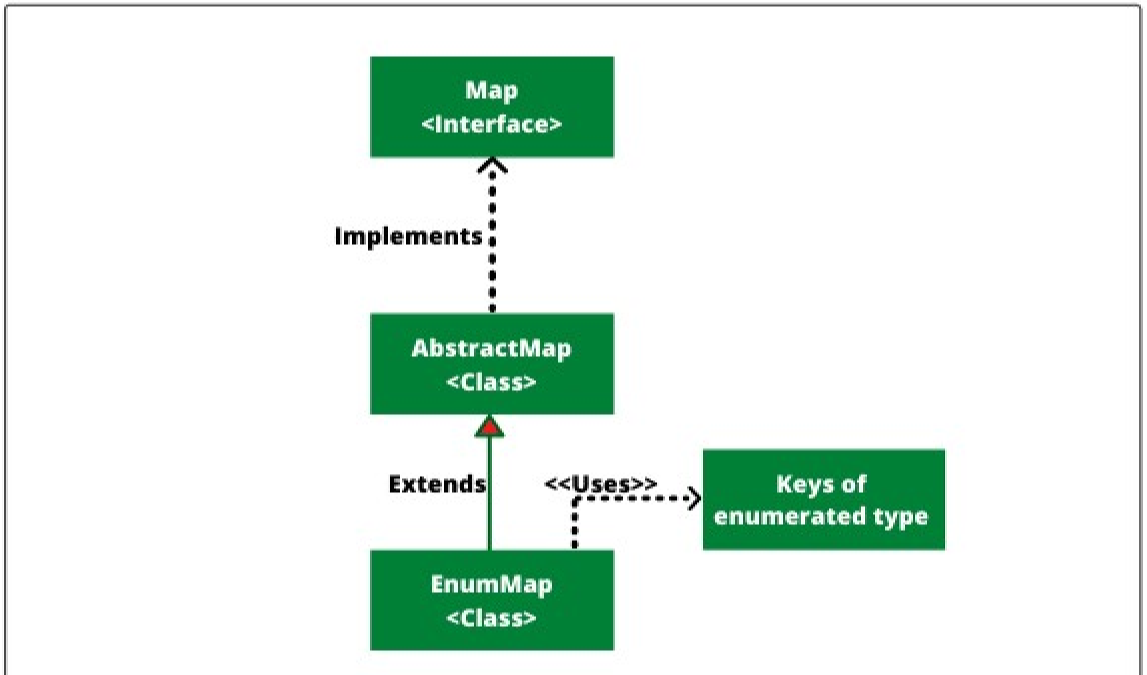


Fig: Hierarchy diagram of EnumMap in Java

## EnumMap class declaration

---

EnumMap is a generic class that is a subclass of the Enum class. Generally, It can be declared as:

```
public class EnumMap<K extends Enum<K>,V>

  extends AbstractMap<K,V>

    implements Serializable, Cloneable
```

EnumMap accepts two parameters as K and V where K defines the type of Keys that extends Enum class and V defines the type of Values.

## Features of EnumMap in Java

There are many important features of enum map in java that are as follows:

1. EnumMap is a Map implementation that is used with keys of enumerated type.

2. Null keys are not permitted in the enum map but null values are permitted. Any attempt to store a null key will throw a named **NullPointerException** because an enum map is represented internally as an array in the terms of performance.
3. Java EnumMap maintains the natural order of their keys in which the enum constants are declared.

4. EnumMap is not synchronized in java. That means it is not thread-safe. Multiple threads can access the same enum map object concurrently.

To synchronize it, use synchronizedMap method of the Collections class. The syntax is given below:

```
Map<EnumKey, V> map = Collections.synchronizedMap(new EnumMap<EnumKey,
V>(...));



For example:

Map<Days, String> map = Collections.synchronizedMap(new EnumMap<Days,
String>(daysEnumMap));
```

5. The iterators returned by the collections views (keySet(), entrySet(), and values()) are weakly consistent. They will never throw **ConcurrentModificationException** and they may or may not show the effects of any modifications to the map that occur while during iteration.

## Constructors of EnumMap class

# Java Map Interface Implementation Classes

Java EnumMap class defines the following constructors. They are as follows:

**1. EnumMap(Class<K> keyType):** This form of constructor creates an empty enum map with the specified keyType. This constructor will throw NullPointerException when keyType is a null reference.
**2. EnumMap(EnumMap em):** This form of constructor constructs an enum map with the same key type as the specified enum map em. It will throw NullPointerException when em contains the null reference.
**3. EnumMap(Map m):** This constructor constructs an enum map that is initialized from the specified map m. If the map m is an enum map then this constructor will behave the same manner as the second constructor.

If the map m is not an enum map, it must have at least one mapping (key-value pairs). It is essential to determine the key type of constructed enum map.

## EnumMap Methods in Java

Following methods are defined by EnumMap class in Java.

**1. void clear():** It removes all the key-value pairs from the map.
**2. EnumMap<K,V> clone():** This method returns a shallow copy of this enum map.
**3. boolean containsKey(Object key):** This method returns true if this map contains a key-value pair for the specified key.
**4. boolean containsValue(Object value):** It returns true if the map maps one or more keys to the specified value.
**5. Set<Map.Entry<K,V>> entrySet():** This method returns a collection or set view of the key-value pairs (mappings) contained in this map.

**6. boolean equals(Object o):** It compares the specified object with this map for equality.
**7. V get(Object key):** It is used to receive the value to which the specified key is mapped, or null if this map contains no mapping for the key.
**8. int hashCode():** It is used to receive the hash code value for this map.
**9. Set<K> keySet():** This method returns a collection view of the keys contained in this map.
**10. V put(K key, V value):** It is used to add the specified value associated with the specified key in this map.
**11. void putAll(Map m):** This method copies all of the key-value pairs from the specified map to this map.
**12. V remove(Object key):** It is used to remove a key-value pair (mapping) for the given key from the map if present.
**13. int size():** It returns the number of key-value mappings in this map.
**14. Collection<V> values():** It returns a set view of the values contained in this map.
**Methods inherited from class AbstractMap:** isEmpty, toString
**Methods inherited from Object class:** finalize, getClass, notify, notifyAll, wait
**Methods inherited from Map interface:** compute, computeIfAbsent, computeIfPresent, forEach, getOrDefault, merge, putIfAbsent, remove, replace, replaceAll

## Java EnumMap Example Programs

---

Let's take an example program based on the methods of enum map in java. Look at the source code.

**Program source code 1:**

```
import java.util.EnumMap;

public class EnumMapEx {
 public enum Days {
     Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday;
}
public static void main(String[] args)
{
// Create a EnumMap.
EnumMap<Days, String> enmap = new EnumMap<Days, String>(Days.class);


// Adding entries in enum map.
 enmap.put(Days.Monday, "Day 2");
 enmap.put(Days.Tuesday, "Day 3");
 enmap.put(Days.Sunday, "Day 1");
 enmap.put(Days.Wednesday, "Day 4");
 enmap.put(Days.Saturday, "Day 7");
 enmap.put(Days.Thursday, "Day 5");
 enmap.put(Days.Friday, "Day 6");


System.out.println("Entries of enum map: " +enmap.entrySet());
System.out.println("Keys of enum map: " +enmap.keySet());
System.out.println("Values of enum map: " +enmap.values());



System.out.println("Is key Sunday present in enum map: "
+enmap.containsKey(Days.Sunday));

System.out.println("Is value Day 3 present in enum map: " +enmap.containsValue("Day
3"));



Object getDay = enmap.get(Days.Saturday);

System.out.println("Value of Saturday: " +getDay);

  }

}

Output:

      Entries of enum map: [Sunday=Day 1, Monday=Day 2, Tuesday=Day 3,
Wednesday=Day 4, Thursday=Day 5, Friday=Day 6, Saturday=Day 7]

      Keys of enum map: [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday]

      Values of enum map: [Day 1, Day 2, Day 3, Day 4, Day 5, Day 6, Day 7]

      Is key Sunday present in enum map: true

      Is value Day 3 present in enum map: true

      Value of Saturday: Day 7
```

**Explanation:**

1. In this program, we have created a class EnumMapEx. Inside the EnumMapEx class, we have created inner class Days of type enum.

2. The EnumMap enmap is constructed inside the main method by using the first constructor and passing the Class of Days enum. The keys are of the type Days. The values are of the type String.

3. The methods entrySet(), keySet(), and values() return a set view of the entries, keys, and a collection view of values, respectively.

4. As you can see in the output of the above program, the ordering of keys is according to the ordering of enum Days.

## How to Iterate EnumMap in Java?

---

In this section, we will iterate over keys, values, and entries of the enum map. We will also remove a key during iteration. Look at the source code to understand better.

**Program source code 2:**

```
import java.util.EnumMap;

import java.util.Iterator;

import java.util.Map.Entry;



public class IteratingEnumMap {



 public enum CartoonCharacter {

    Archie, Sabrina, Tom, Jerry, Mickey, Richie;

 }

public static void main(String[] args)
```

```
{



EnumMap<CartoonCharacter, String> enmap = new EnumMap<CartoonCharacter,
String>(CartoonCharacter.class);



 enmap.put(CartoonCharacter.Archie, "Main hero");

 enmap.put(CartoonCharacter.Sabrina, "Teenager");

 enmap.put(CartoonCharacter.Jerry, "Mouse");

 enmap.put(CartoonCharacter.Tom, "Cat");

 enmap.put(CartoonCharacter.Richie, "Richest kid");

 enmap.put(CartoonCharacter.Mickey, "Mouse");



System.out.println("Iterating keys of enum map");

Iterator<CartoonCharacter> keySet = enmap.keySet().iterator();



while(keySet.hasNext()){

  Object key = keySet.next();

  System.out.println(key);

}

System.out.println("\n");
System.out.println("Iterating values in enum map");

Iterator<String> values = enmap.values().iterator();



while(values.hasNext()){

   System.out.println(values.next());

}

System.out.println("\n");



System.out.println("Iterating entries in enum map");

Iterator<Entry<CartoonCharacter,String>> entrySet = enmap.entrySet().iterator();



while(entrySet.hasNext()){

  Object eSet = entrySet.next();

  System.out.println(eSet);

}

// Call remove() method to remove entry for specified key from enum map and returns
value associated with specified key.

  System.out.println("Value associated with removing key: "
+enmap.remove(CartoonCharacter.Mickey));

  }
```

```
}

Output:

    Iterating keys of enum map

    Archie

    Sabrina

    Tom

    Jerry

    Mickey

    Richie



    Iterating values in enum map

    Main hero

    Teenager

    Cat

    Mouse

    Mouse

    Richest kid



    Iterating entries in enum map

    Archie=Main hero
```

```
Sabrina=Teenager

Tom=Cat

Jerry=Mouse

Mickey=Mouse

Richie=Richest kid

Value associated with removing key: Mouse
```

## SortedMap in Java | Methods, Example

**SortedMap in Java** is an interface that is a subinterface of Map interface. The entries in the map are maintained in sorted ascending order based on their keys.
In simple words, it guarantees that the entries are maintained in ascending key order. It allows very efficient manipulations of subsets of a map.

SortedMap was added in Java 1.2 version. It is present in java.util.SortedMap package.

### Hierarchy of SortedMap Interface in Java

SortedMap interface extends Map interface. TreeMap and ConcurrentSkipListMap classes implements the SortedMap interface. The hierarchy diagram of SortedMap interface in java is shown below in the figure.

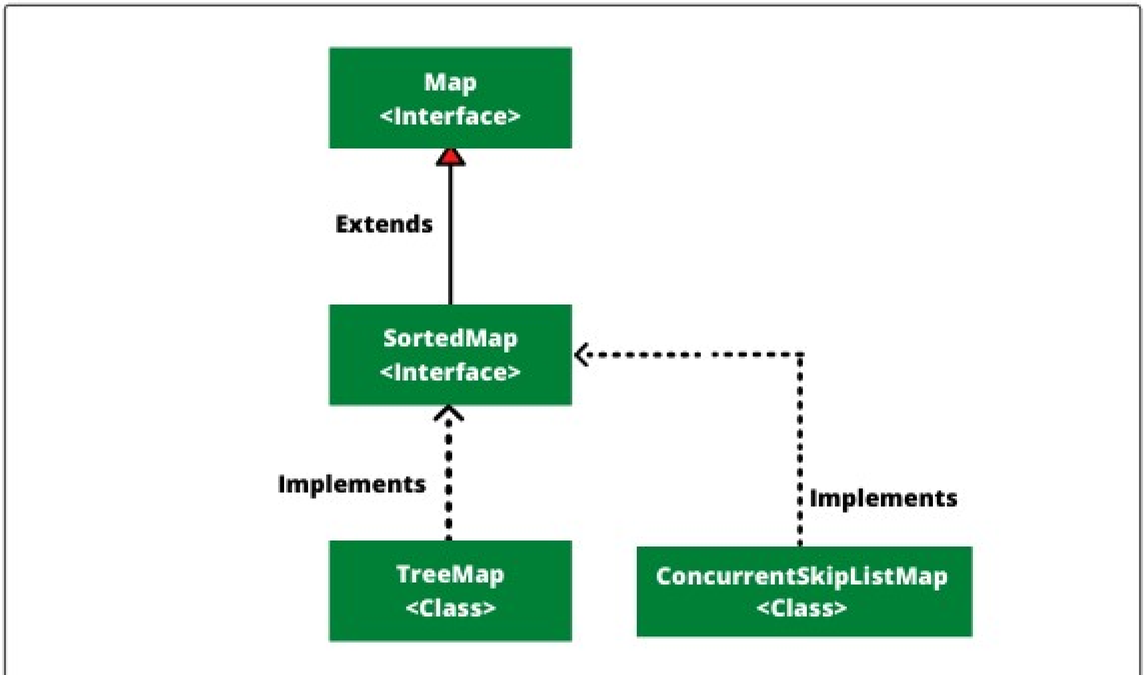


Fig: Hierarchy diagram of SortedMap in Java

## SortedMap Interface

---

SortedMap is a generic interface that is declared as shown below:

```
interface SortedMap<K,V>

    extends Map<K,V>
```

Here, K represents the type of keys maintained by this map and V represents the type of mapped values.

## SortedMap Methods in Java

---

# Java Map Interface Implementation Classes

In addition to the methods provided by Map interface, SortedMap also provides several additional methods that are as follows:

**1. Comparator comparator():** This method returns the comparator used to order the keys in this map. If the map uses the natural ordering of its keys, null is returned.
**2. Set<Map.Entry<K,V>> entrySet():** This method returns a set view of the key-value pairs (mappings) contained in this map.
**3. Set<K> keySet():** It returns a set view of the keys contained in this map.
**4. K firstKey():** It returns the first (lowest or smallest) key in the invoking map.
**5. K lastKey():** It returns the last (highest or largest) key in the invoking map.
**6. SortedMap<K,V> headMap(K toKey):** This method returns a portion of the map whose keys are strictly less than toKey.
**7. SortedMap<K,V> tailMap(K fromKey):** This method returns a portion of the map whose keys are greater than or equal to fromKey.

**8. SortedMap<K,V> subMap(K fromKey, K toKey):** This method returns a portion of the map whose keys range from fromKey, inclusive, to toKey, exclusive.

**Methods inherited from Map interface**
clear(), containsKey(), containsValue(), equals(), get(), hashCode(), isEmpty(), put(), putAll(), remove(), size()

Several methods throw an exception named NoSuchElementException if no entries are in the invoking map. When an object is not compatible with elements in the map, ClassCastException is thrown.

If you attempt to use a null object when null is not allowed in the map, a runtime exception named NullPointerException will be thrown.

## Java SortedMap Example Programs

# Java Map Interface Implementation Classes

Let’s take example programs for performing the various operations based on the methods defined by SortedMap interface in Java.

Since Java SortedMap is an interface, it can be used only with class that implements SortedMap interface. TreeMap class in Java implements the SortedMap interface.

**1. Adding and Removing Elements:** Let’s create a program where we will add and remove an element to the SortedMap using the put() and remove() methods respectively. However, the insertion order is not preserved in the TreeMap.
Internally, for every element, the keys are compared and sorted according to ascending order. Look at the following source code to understand better.

**Program source code 1:**

```
import java.util.SortedMap;

import java.util.TreeMap;

public class SortedMapEx {

public static void main(String[] args)

{

// Create a SortedMap.

  SortedMap<Integer, String> smap = new TreeMap<>();



// Adding entries in TreeMap.

   smap.put(10, "Ten");
   smap.put(20, "Twenty");

   smap.put(05, "Five");

   smap.put(07, "Seven");

   smap.put(40, "Forty");



System.out.println("Entries in map: " +smap);



// Removing an entry.

   Object removeEntry = smap.remove(05);

   System.out.println("Removed entry: " +removeEntry);

   System.out.println("Updated entries in map after remove operation: " +smap);



  }

}

Output:

    Entries in map: {5=Five, 7=Seven, 10=Ten, 20=Twenty, 40=Forty}

    Removed entry: Five

    Updated entries in map after remove operation: {7=Seven, 10=Ten, 20=Twenty,
40=Forty}
```

**2. Updating entries:** After adding entries if you want to update an entry, it can be done by again adding an entry with the put() method.

Since entries in the SortedMap are indexed using keys, the value of the key can be updated or changed by simply inserting the updated value for the key for which you wish to change.

**Program source code 2:**

```
import java.util.SortedMap;

import java.util.TreeMap;

public class SortedMapEx2 {

public static void main(String[] args)

{

// Create a SortedMap.

  SortedMap<Integer, String> smap = new TreeMap<>();



   smap.put(06, "Six");

   smap.put(20, "Twenty");

   smap.put(05, "Fie"); // Here, an entry is wrong.

   smap.put(07, "Seven");

   smap.put(40, "Forty");



System.out.println("Entries in map: " +smap.entrySet());



// Updating the spelling of wrong entry.
```

```
  smap.put(05, "Five");

  System.out.println("Updated entries in map: " +smap.entrySet());

  System.out.println("SubMap: " +smap.subMap(10, 45));

 }

}

Output:

      Entries in map: [5=Fie, 6=Six, 7=Seven, 20=Twenty, 40=Forty]

      Updated entries in map: [5=Five, 6=Six, 7=Seven, 20=Twenty, 40=Forty]

      SubMap: {20=Twenty, 40=Forty}
```

3. Let’s take another example program to perform various operations based on headMap(), tailMap(), firstKey(), and lastKey() methods. Look at the following source code to understand better.

**Program source code 3:**

```
import java.util.SortedMap;

import java.util.TreeMap;

public class SortedMapEx3 {

public static void main(String[] args)

{

// Create a SortedMap.

  SortedMap<Integer, String> smap = new TreeMap<Integer,String>();
```

## Java Map Interface Implementation Classes

```
// Adding entries in TreeMap.

   smap.put(90, "John");

   smap.put(85, "Deep");

   smap.put(100, "Sophia");

   smap.put(35, "Olivea");

   smap.put(39, "Shubh");


System.out.println("Marks in maths: " +smap);

System.out.println("Students that passed maths exam: " +smap.tailMap(40));

System.out.println("Students that failed in maths exam: " +smap.headMap(40));


System.out.println("Lowest marks: " +smap.firstKey());

System.out.println("Highest marks: " +smap.lastKey());

 }

}

Output:

      Marks in maths: {35=Olivea, 39=Shubh, 85=Deep, 90=John, 100=Sophia}

      Students that passed maths exam: {85=Deep, 90=John, 100=Sophia}

      Students that failed in maths exam: {35=Olivea, 39=Shubh}

      Lowest marks: 35
```

## NavigableMap in Java | Methods, Example

**NavigableMap in Java** is an interface that is a subinterface of SortedMap interface.
It extends SortedMap interface to handle the retrieval of entries based on the closet match to a given key or keys.

NavigableMap interface was added by Java 1.6 version. It is present in java.util.NavigableMap package. NavigableMap interface is recently added in Java Collections Framework.

TreeMap class is a widely used class that implements NavigableMap interface in Java. NavigableMap interface provides several methods to make map navigation easy.
In simple words, we can navigate the Map easily with NavigationMap interface. We can retrieve the nearest value matching with the specified key, all the values less than the specified key, all values greater than the given key, and so on.

### Hierarchy of NavigableMap interface in Java

NavigableMap interface extends SortedMap interface and Map interface. Java ConcurrentNavigableMap interface is the subinterface of NavigableMap interface.

The hierarchy diagram of NavigableMap interface is shown below in the figure.

## Java Map Interface Implementation Classes

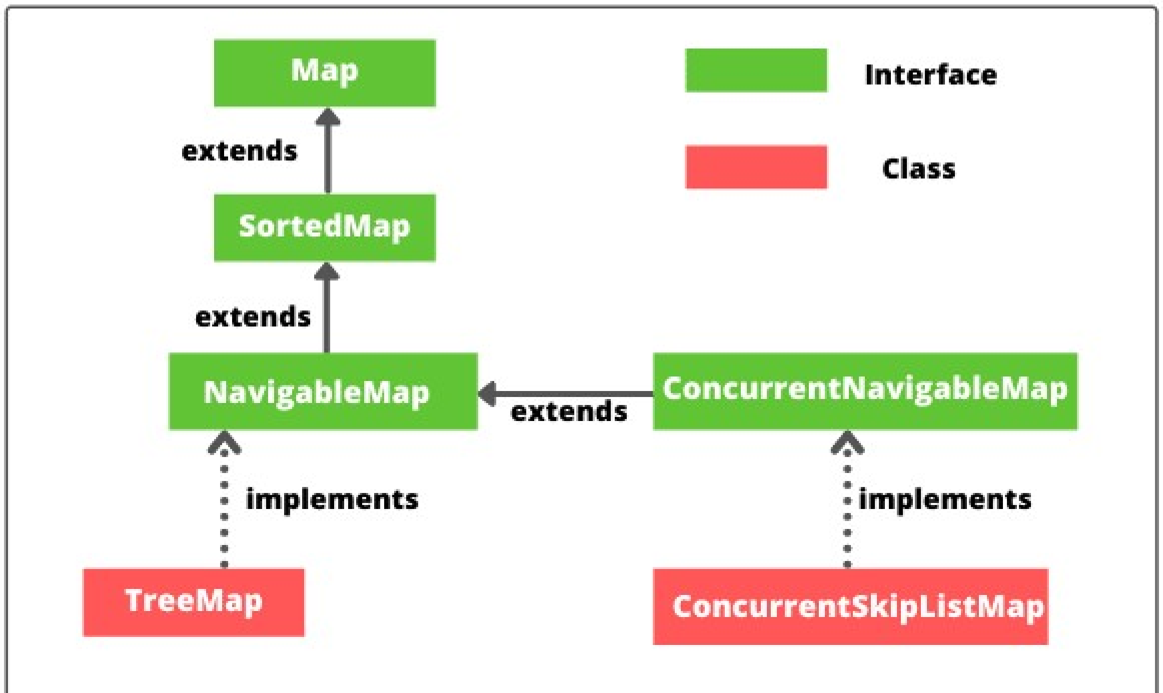


Fig: Java NavigableMap Hierarchy diagram

### NavigableMap Interface declaration

---

NavigableMap is a generic interface that is declared as below:

```
public interface NavigableMap<K,V>

    extends SortedMap<K,V>
```

Here, K represents the type of keys and V represents the type of values corresponding to keys.

### NavigableMap Methods in Java

---

In addition to the methods that it inherits from the SortedMap interface, NavigableMap interface in Java also adds many new methods. They are as follows:

**1. Map.Entry<K,V> ceilingEntry(K obj):** This method searches the map for the smallest (least) key k such that key k is greater than or equal to the given key obj (k >= obj).
If such key is found, its entry is returned. If not found such key, the null object will be returned.

**2. K ceilingKey(K obj):** This method searches the map for the smallest key k such that key k is greater than or equal to the given key obj (k >= obj).
If such key is found, it is returned. If not found such key, the null object will be returned.

**3. NavigableSet<K> descendingKeySet():** This method returns a NavigableSet view that contains keys in the invoking map in reverse order. In simple words, it returns a reverse set view of the keys.
**4. NavigableMap<K,V> descendingMap():** It returns NavigableMap that is a reverse of invoking map.
**5. Map.Entry<K,V> firstEntry():** It returns the first key-value pair associated with the least key in the map, or null if the map is empty.
**6. Map.Entry<K,V> floorEntry(K obj):** This method searches the largest key k such that the key k is less than or equal to specified key obj (k <= obj).
If such a key is found in the map, its entry is returned. Otherwise, null object is returned.

**7. K floorKey(K obj):** This method searches the largest key k such that the key k is less than or equal to specified key obj (k <= obj).
If such a key is found in the map, it is returned. Otherwise, the null object is returned.

**8. NavigableMap<K,V> headMap(K toKey, boolean inclusive):** This method returns a portion of NavigableMap that includes all entries from

the invoking map whose keys are less than toKey. If inclusive is true, an element equal to toKey is included.
**9. Map.Entry<K,V> higherEntry(K obj):** It searches a set for the largest key k such that k > obj. If such a key is found in the map, its key-value pair is returned. Otherwise, the null object is returned.
**10. K higherKey(K obj):** It searches a set for the largest key k such that k > obj. If a such key is found in the map, it is returned. Otherwise, null object is returned.
**11. Map.Entry<K,V> lastEntry():** This method returns the last entry associated with the largest key in the map, or null if the map is empty.
**12. Map.Entry<K,V> lowerEntry(K obj):** This method searches the set for largest key k such that k < obj. If the key is found in the map, its entry is returned. Otherwise, null object is returned.
**13. K lowerKey(K obj):** This method searches the set for largest key k such that k < obj. If the key is found in the map, it is returned. Otherwise, null object is returned.
**14. NavigableSet<K> navigableKeySet():** It returns a NavigableSet view of the keys contained in the invoking map.

## Comparable Interface in Java | Use, Example

**Comparable interface in Java** defines compareTo() method for comparing objects of the same type.
In other words, Comparable interface defines a standard way in which two objects of the same class will be compared.

It is mainly used to provide the natural sorting order of objects (elements). Natural ordering means usual ordering.

For example, the natural order of strings is alphabetical (i.e. A before B, B before C, and so on). For numeric values, it is numeric order (i.e. 1 before 2, 2 before 3, and so on).

Java Comparable interface can be implemented by any custom class or user-defined class if you want to use Arrays or Collections sorting methods.

# Java Map Interface Implementation Classes

The sorted collections use the natural sorting order defined by a class to determine the order of objects. So, if you want to store objects in a sorted collection, its class must implement Comparable interface.

Comparable interface was introduced in Java 1.0 version. It is present in java.lang.Comparable package and contains only one method named compareTo(Object).

## When to use Comparable interface in Java?

---

Suppose we want to find the larger or smaller of two objects of the same type, such as two students, two employees, two dates, two rectangles, two squares, or two circles.

In order to compare it, the two objects must be comparable. To accomplish this purpose, Java provides Comparable interface.

The purpose of using Comparable interface and its compareTo() method is to compare one object with another object in a way less than (<), equal to (=), or greater than (>).

Look at the below realtime example to understand better.

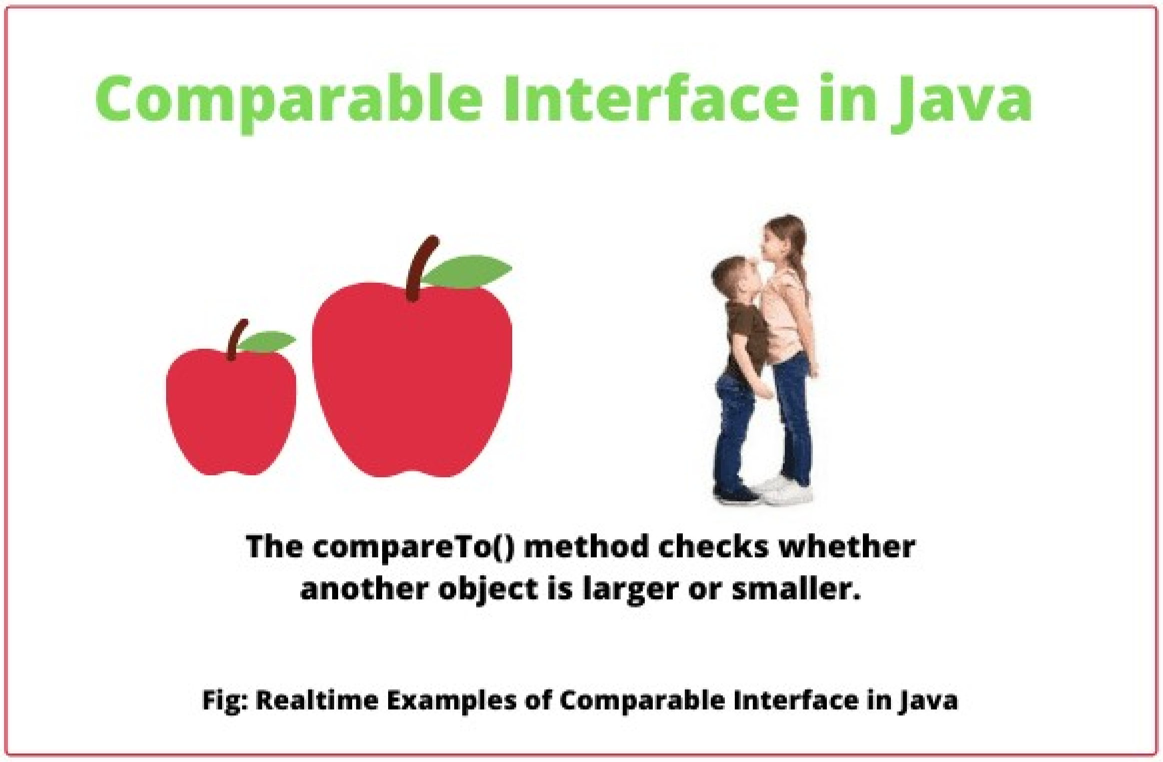

Comparable Interface in Java
The compareTo() method checks whether another object is larger or smaller.

Fig: Realtime Examples of Comparable Interface in Java

## Comparable Interface declaration

---

Comparable in Java is a generic interface that can be declared like this in the general form:

```
public interface Comparable<T>
```

Here, T represents the type of objects that has to be compared. Several classes in Java library such as BigDecimal, BigInteger, Boolean, Byte, ByteBuffer, Calendar, String, etc implement Comparable interface in java to define a natural order for objects.

## compareTo(Object obj) Method in Java

# Java Map Interface Implementation Classes

The Comparable interface provides only one abstract method that is used to determine the natural ordering of instances of a class. The signature of this method is as follows:

```
public int compareTo(Object obj)



// This method is defined in Comparable interface for comparing objects (defined in
java.lang package) like this:



 package java.lang;

  public interface Comparable<E> {

      public int compareTo(Object obj);

  }
```

The compareTo() method is used to compare the current object with the specified object. It returns an indication that is as follows:

- positive integer value, if the current object is greater than the specified object (this > object).
- negative integer value, if the current object is less than the specified object (this < object).
- zero, if the current object is equal to the specified object (this = object).

Let's understand it with the following examples.

1. a.compareTo(b) must return a negative number if a should come before b, zero if a and b are the same, and a positive number if b should come before a.

2. System.out.println(new Integer(3).compareTo(new Integer(5))); must return a negative value becuase 3 is less than 5.

3. System.out.println("ABC".compareTo("ABE")); must retrun negative value becaue ABC is less than ABE.

4. java.util.Date date1 = new java.util.Date(2021, 1, 1);
java.util.Date date2 = new java.util.Date(2020, 1, 1);
System.out.println(date1.compareTo(date2)); must return a positive value because date1 is greater than date2.

Thus, numbers, strings, dates are comparable. We can use the compareTo() method to compare two numbers, two strings, and two dates.

In general, we can sort elements of String objects, Wrapper class objects, and User-defined class objects.

If the two objects are not compatible with each other, the compareTo() method will throw an exception named **ClassCastException**.

## How to Implement Comparable Interface in Java with Example Programs

---

Several classes in the Java library implement Comparable interface to define a natural sorting order for objects. These classes are Byte, Short, Integer, Long, Float, Double, Character, BigInteger, BigDecimal, Calendar, String, Date, etc.

All these classes implement the Comparable interface. For example, the String, and Integer classes are defined as follows in the Java API:

```
1. public class String extends Object

       implements Comparable<String> {
```

```
    // class body omitted
   @Override
   public int compareTo(String o) {
     // Implementation omitted
     }
   }
2. public class Integer extends Number
        implements Comparable<Integer> {
     // class body omitted
  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
   }
  }
```

**Note:** String and Wrapper classes implement Comparable interface by default.

1. Let's take an example program where we will define a user-defined class Student that will implement Comparable interface to sort the list of elements on the basis of roll numbers. Look at the program source code to understand better.

**Program source code 1:**

```
public class Student implements Comparable<Student> {
```

```
// Declaration of variables.

    String name;

    int rollno;

    int age;


Student(String name, int rollno,int age){

    this.name = name;

    this.rollno = rollno;

    this.age = age;

  }

// Compare two students based on their roll numbers.


@Override // Implementing the compareTo method defined in Comparable interface.

public int compareTo(Student st)

{

// Logic for sorting elements in ascending order.

 if(rollno == st.rollno)

  return 0;

             else if(rollno > st.rollno)

             return 1;
```

```
            else

            return -1;

   }

}

import java.util.ArrayList;

import java.util.Collections;

public class TestNaturalOrder {

public static void main(String[] args)

{

// Creates objects of Student class and passes the parameters to their constructors.

   Student st1 = new Student("John", 20, 15);

   Student st2 = new Student("Peter", 15, 16);

   Student st3 = new Student("Deep", 25, 15);


// Create an object of ArrayList of type Student.

   ArrayList<Student> al = new ArrayList<>();


// Adds elements (references) to the array list.

   al.add(st1);

   al.add(st2);
```

```
  al.add(st3);


// Display name of students, sorted by rollnos.
 System.out.println("Displaying student's name sorted by rollnos:");
 Collections.sort(al); // This method is used to sort elements of list.
 for(Student st:al){
      System.out.println(st.name+" "+st.rollno+" "+st.age);
  }
 }
}
Output:
     Displaying student's name sorted by rollnos:
     Peter 15 16
     John 20 15
     Deep 25 15
```

2. Let's take the same example program where we will sort the list of elements on the basis of rollnos in reverse order. We only need to change the particular code in the Student class.

**Program source code 2:**

```
public int compareTo(Student st)
{
```

```
// Logic for sorting elements in descending order or reverse order.

 if(rollno == st.rollno)

  return 0;

            else if(rollno < st.rollno)

            return 1;

            else

            return -1;

 }
```

3. Let’s take one more example program where we will learn how to sort employee objects in java using comparable interface. This example program will clear all doubts of the comparable interface.

In this example program, we will create a class Employee that will implement Comparable interface and will sort employee information based on their hike salary.

Look at the following source code to understand better.

**Program source code 3:**

```
public class Employee implements Comparable<Employee> {

// Declaration of encapsulated variables.

  private String name;

  private double salary;



public Employee(String name, double salary){
  this.name = name;

  this.salary = salary;

  }

public String getName() {

   return name;

 }

public double getSalary() {

   return salary;

 }

public void hikeSalary(double byPercent){

  double hike = salary * byPercent / 100;

   salary += hike;

 }

@Override

public int compareTo(Employee emp)

{

 if (salary < emp.salary) return -1;

 if (salary > emp.salary) return 1;

   return 0;

  }
```

```
}

import java.util.Arrays;

public class EmployeeSortTest {

public static void main(String[] args)

{

// Create an array.

Employee[] staff = new Employee[3];

            staff[0] = new Employee("Harry", 30000);

            staff[1] = new Employee("Carl", 70000);

            staff[2] = new Employee("Tony", 39000);


  Arrays.sort(staff); //  This method is used to sort list of elements in Arrays.



// Display information about all Employees, sorted by their salaries.

  for (Employee e : staff)

     System.out.println("name: " + e.getName() +", "+"salary = " + e.getSalary());

    }

}

Output:

     name: Harry, salary = 30000.0
```

```
name: Tony, salary = 39000.0

name: Carl, salary = 70000.0
```

## Comparator in Java | Use, Example

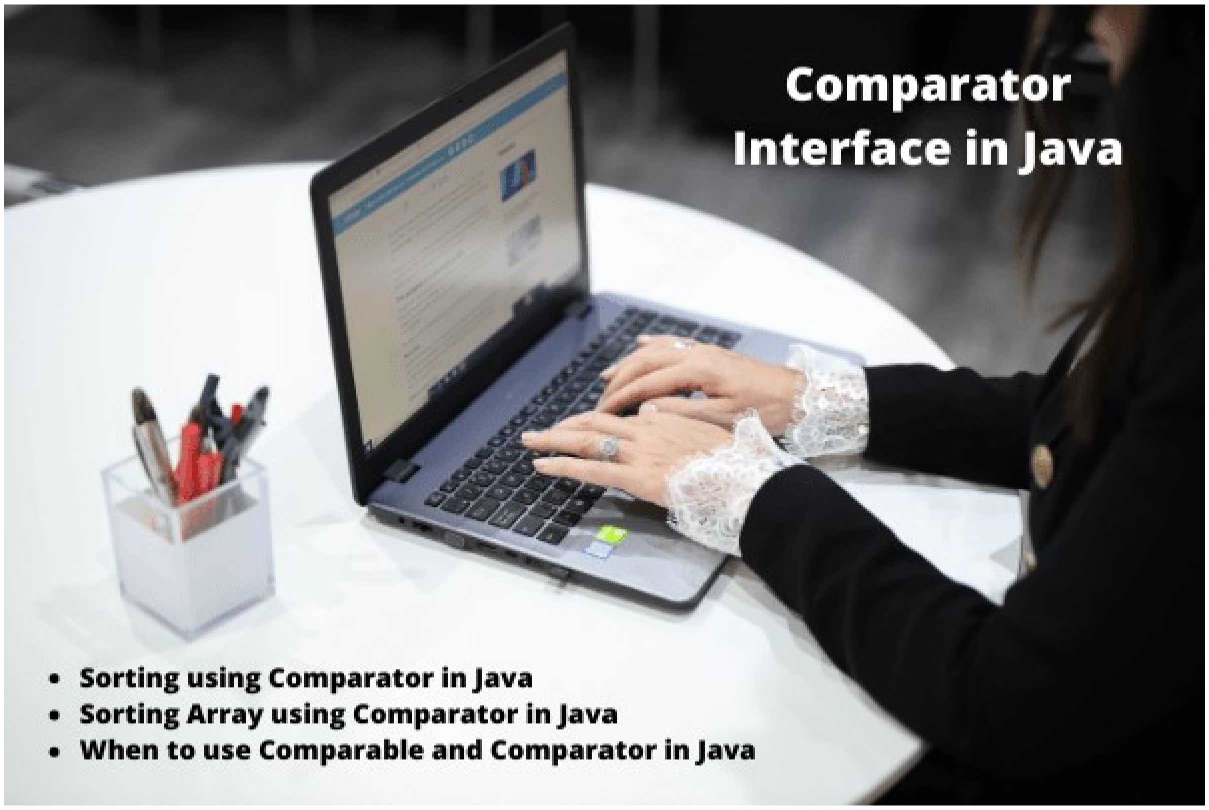
**Comparator in Java** is an interface whose task is to compare objects of a user-defined class when the class does not implement the Comparable interface.
In other words, Java Comparator is used when:

- Objects to be ordered do not have a natural ordering defined by Comparable interface.
- You want to order elements something different way other than their natural order.

In both cases, we can specify a Comparator object when you construct the set or map.

Comparator interface was introduced in Java 1.2 version. It is present in java.util.Comparator<T> package.

# Java Map Interface Implementation Classes


Comparator Interface in Java
Sorting using Comparator in Java
Sorting Array using Comparator in Java
When to use Comparable and Comparator in Java

## Comparator Interface declaration

---

Comparator is a generic interface that can be declared in general form like this:

```
Interface Comparator<T>
```

Here, the parameter T represents the type of objects (or elements) that will be compared.

## Methods of Java Comparator Interface

---

Comparator interface provides two methods that are as follows:

## Java Map Interface Implementation Classes

**1. int compare(Object obj1, Object obj2):** It compares the first object obj1 with the second object obj2. It returns zero if objects are equal (i.e. obj1 == obj2).
It returns a positive value if obj1 is greater than obj2 (i.e. obj1 > obj2). Otherwise, a negative value is returned.

Let's take an example to understand better. Comparator<Country> looks like this:

```
public interface Comparator<Country>

{

  int compare(Country a, Country b);

}
```

The call to com.compare(a, b); will return a negative value if a come before b, 0 if a and b are equal, and a positive value. Here, com is an object of a class that implements Comparator<Country>.

The compare() method can throw an exception named ClassCastException if types of objects are not compatible for comparison.

By overriding the compare( ) method, we can change the way that objects are ordered. For example, to sort elements in reverse order, we can create a comparator object that reverses the outcome of a comparison.

**2. boolean equals(Object obj):** The equals() method defined in the Comparator interface is overridden of the equals() method of the Object class. It is used to test whether an object equals invoking comparator. Here, the parameter obj is the object to be tested for equality. If object obj and the invoking object both are Comparator objects and use the same ordering. the equals() method returns true. Otherwise, it returns false.

Overriding equals( ) is not always necessary. When it is not necessary, there is no requirement to override Object's implementation.

The complete declaration of the Comparator class that is defined in java.util package is as follows:

```
public interface Comparator<T>

{

 // An abstract method declared in the interface

      int compare(Object obj1, Object obj2);



// Re-declaration of the equals() method of the Object class

      boolean equals(Object obj);

}
```

## Example Programs based on Sorting using Comparator in Java

Let's take an example program where we will perform sorting of integer elements in ascending order using comparator. Look at the following source code to understand the power of custom comparator.

**Program source code 1:**

```
import java.util.Comparator;

// To sort elements into ascending order.

public class Ascend implements Comparator<Integer>

{
```

```
@Override

public int compare(Integer i1, Integer i2) {

   return i1.compareTo(i2);

  }

  // No need to override equals method.

}

import java.util.TreeSet;

public class CompTest {

public static void main(String[] args)

{

// Create an object of Ascend class.

   Ascend as = new Ascend();



// Create a tree set and pass the reference variable of Ascend class as a parameter.

  TreeSet<Integer> ts = new TreeSet<Integer>(as);



// Adds elements into treeset.

  ts.add(25);

  ts.add(15);

  ts.add(30);
  ts.add(10);

  ts.add(40);

  ts.add(05);



// Display the elements in ascending order.

  System.out.println("Sorted in Ascending order");



 for(Integer element : ts)

    System.out.print(element + " ");

    System.out.println();

   }

}

Output:

      Sorted in Ascending order

      5 10 15 25 30 40
```

**Explanation:** Look closely at the Ascend class, which implements Comparator and overrides compare( ) method. As explained earlier, overriding equals( ) method is needed here. Inside compare( ) method, the compareTo( ) compares the two integers.

## How to Sort Elements in Descending Order using Comparator in Java

Let's write a program to sort elements in descending order using Comparator in Java. Look at the source code.

**Program source code 2:**

```
import java.util.Comparator;

// To sort elements into descending order.

public class Descend implements Comparator<Integer>{

@Override // Implement compare() method to reverse for the integer comparison.

public int compare(Integer i1, Integer i2)

{

// For reverse comparison.

   return i2.compareTo(i1);

 }

}

import java.util.TreeSet;

public class CompTest {

public static void main(String[] args)

{

// Create an object of Descend class.

  Descend ds = new Descend();
// Create a tree set and pass the reference variable of Descend class as a parameter.

  TreeSet<Integer> ts = new TreeSet<Integer>(ds);



// Adds elements into tree set.

  ts.add(25);

  ts.add(15);

  ts.add(30);

  ts.add(10);

  ts.add(40);

  ts.add(05);



// Display the elements in ascending order.

  System.out.println("Sorted in Descending order");



for(Integer element : ts)

    System.out.print(element + " ");

    System.out.println();

   }

}
```

```
Output:

     Sorted in Descending order

     40 30 25 15 10 5
```

Let’s take one more example program similar to the above program where we will perform reverse string comparison.

**Program source code 3:**

```
import java.util.Comparator;

// To sort elements into descending order.

public class RevStrComp implements Comparator<String>

{

@Override // Implements compare() method for reverse string comparison.

public int compare(String str1, String str2)

{

// For reverse comparison.

   return str2.compareTo(str1);

 }

}

import java.util.TreeSet;

public class CompTest {

public static void main(String[] args)

{
```

```
// Create an object of RevStrComp class.

  RevStrComp rsc = new RevStrComp();



// Create a tree set and pass reference variable of RevStrComp class as parameter.

  TreeSet<String> ts = new TreeSet<String>(rsc);



// Adds elements into tree set.

  ts.add("Cat");

  ts.add("Elephant");

  ts.add("Lion");

  ts.add("Dog");

  ts.add("Tiger");

  ts.add("Horse");



// Display the elements in ascending order.

  System.out.println("Sorted in reverse order");



for(String element : ts)

    System.out.print(element + " ");

    System.out.println();
  }

}

Output:

     Sorted in reverse order

     Tiger Lion Horse Elephant Dog Cat
```

## Sorting Array using Comparator

---

Let’s take an example program where we will sort an array with group of integer elements (objects) using Comparator in java.

Here, we will accept array elements from the keyboard and sort them in ascending and descending order. Look at the following source code.

**Program source code 4:**

```
import java.util.Comparator;

// To sort elements into ascending order.

public class Ascend implements Comparator<Integer>

{

@Override

public int compare(Integer i1, Integer i2) {

            return i1.compareTo(i2);

 }

}
```

```
import java.util.Comparator;

// To sort elements into descending order.

public class Descend implements Comparator<Integer>{

@Override

public int compare(Integer i1, Integer i2)

{

// For reverse comparison.

  return i2.compareTo(i1);

 }

}

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Arrays;

public class ArraysCompTest {

public static void main(String[] args) throws NumberFormatException, IOException

{

// Create an object of InputStreamReader class to accept array elements from the
keyboard.

 InputStreamReader isr = new InputStreamReader(System.in);
```

```
// Create an object of BufferedReader and pass the reference variable to its constructor.
   BufferedReader br = new BufferedReader(isr);

 System.out.println("How many elements do you want to enter?");
 int size = Integer.parseInt(br.readLine());

// Create an array to store Integer type elements or objects.
  Integer arr[] = new Integer[size];

// Now convert int values into Integer objects and then pass to array to store them.
   for(int i = 0; i < size; i++){
              System.out.println("Enter your number:");
              arr[i] = Integer.parseInt(br.readLine());
   }
// Create an object of Ascend class.
   Ascend as = new Ascend();

// Call sort() method to sort array elements in ascending order.
   Arrays.sort(arr, as); //  for sorting into ascending order.
```

```
// Display the sorted array elements.
   System.out.println("\nSorted in Ascending order: ");
   display(arr);

// Create an object of Descend class.
   Descend ds = new Descend();

// Call sort() method to sort array elements in descending order.
   Arrays.sort(arr, ds); // for sorting into descending order.

// Display the sorted array elements.
   System.out.println("\nSorted in Descending order: ");
   display(arr);
  }
static void display(Integer arr[]){
 for(Integer element : arr){
             System.out.println(element + "\t");
   }
  }
```

```
}

Output:

    How many elements do you want to enter?

    5

    Enter your number:

    12

    Enter your number:

    10

    Enter your number:

    15

    Enter your number:

    20

    Enter your number:

    05


    Sorted in Ascending order:

    5

    10

    12

    15
```

```
20


Sorted in Descending order:

20

15

12

10

5
```

## When to use Comparable and Comparator interfaces in Java

Both Comparable and Comparator interfaces are used to sort the collection or array of elements (objects). But there are the following differences in use. They are as follows:

**Use of Comparable interface in Java**

1. Comparable interface is used for the natural sorting of objects. For example, if we want to sort Employee class by employeeId is natural soring.

2. Comparable is used to sort collection of elements based on only a single element (or logic) like name.

3. Comparable interface is used when we want to compare itself with another object.

**Use of Comparator interface in Java**

## Java Map Interface Implementation Classes

1. Comparator interface is used when we want to perform custom sorting on the collection of elements. It gives exact control over the ordering. For example, if we want to sort Employee class by name and age, it will be custom sorting.

2. Comparator is used to sort collection of elements based on multiple elements (or logics )like name, age, class, etc.

3. Comparator interface is used when we want to compare two different objects.

## Queue in Java | Methods, Example

A **Queue in Java** is a collection of elements that implements the "**First-In-First-Out**" order.
In simple words, a queue represents the arrangement of elements in the first in first out (FIFO) fashion.

That means an element that is stored as a first element into the queue will be removed first from the queue. That is, the first element is removed first and last element is removed at last.

Java Collection Framework added Queue interface in Java 5.0. It is present in java.util.Queue

### Realtime Example of Queue

1. The most common realtime example of Queue is a waiting line in the supermarket. The cashier services the person that is at the beginning of the line first. Other customers enter the line only at the other end and wait for service.

A queue in the super market

Similarly, a queue in java is a common data structure in which there are two ends: front (head) and back (tail) of the queue. Elements are always inserted into the end (tail) of the queue and are accessed and deleted from the beginning (head) of the queue, as shown in the below figure.

For this reason, a queue is a **first-in, first-out (FIFO) data structure**. The operations of inserting and removing elements are called **enqueue** and **dequeue**.

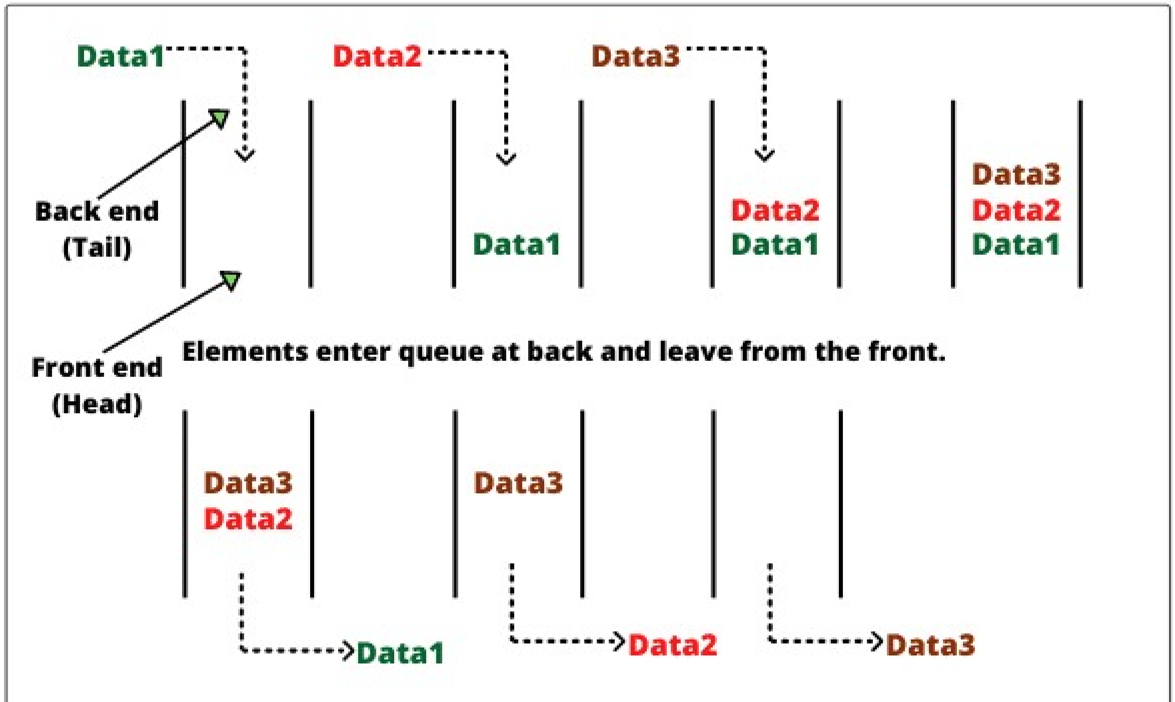


Fig: A queue holds elements in a first-in, first-out fashion

An element that is added to the end of the queue will remain on the queue until all the elements in front of it have been removed.

This process is similar to the "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive in the queue.

## Hierarchy of Queue Interface in Java

---

In the Java Collections Framework, a queue is represented by java.util.queue interface. Java Queue interface extends the Collection interface. It is the super interface of BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>.

A Queue interface is implemented by several classes such as LinkedList, AbstractQueue, ArrayBlockingQueue, ArrayDeque, and PriorityQueue. The hierarchy diagram of Queue interface in Java is shown below in the figure.

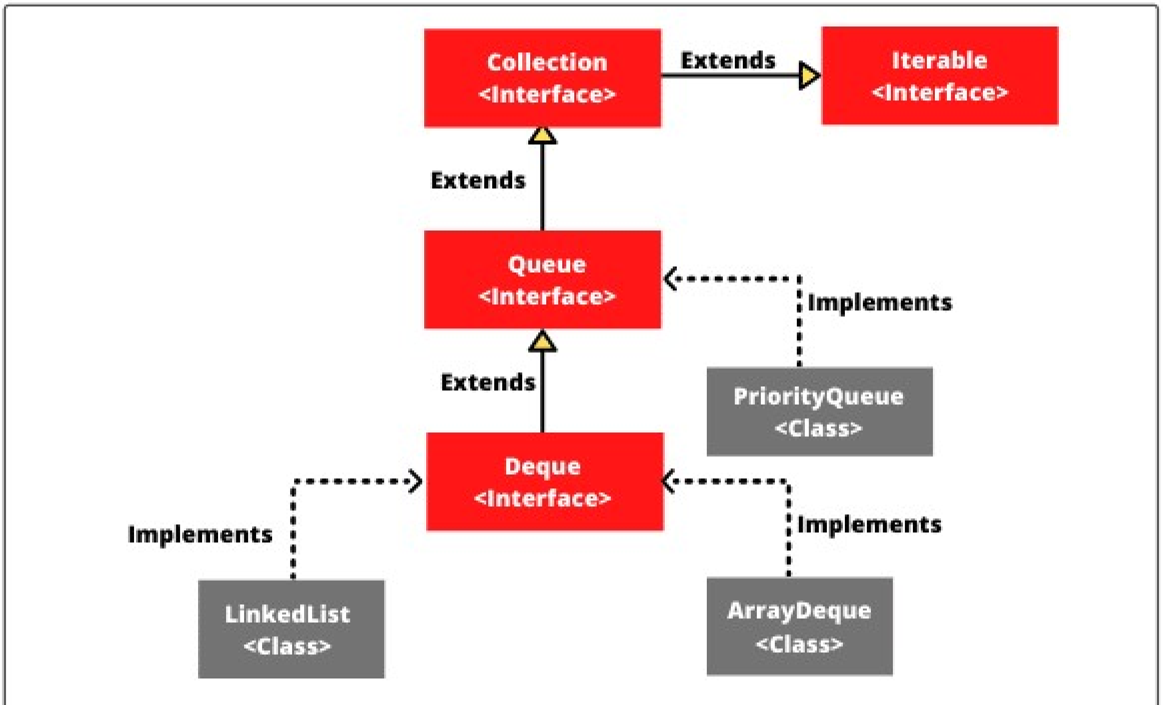


Fig: Hierarchy diagram of Queue in Java

Out of these, ArrayDeque class is the best choice for simple FIFO queues. It provides all the normal functionalities of a queue.

## Queue Interface declaration

---

Queue is a generic interface that is declared in a general form as below:

```
public interface Queue<E>

    extends Collection<E>
```

In the above syntax, E represents the type of objects that the set will hold.

## Features of Java Queue interface

---

There are several interesting features of the queue in Java that is as follows:

1. Java Queue interface orders elements in First In First Out policy.
2. Elements can be accessed and removed only from the front (head) of the queue.
3. Elements can be added only from the back (tail) of the queue.
4. Queue does not allow to add the null object.

## How to create a Queue in Java?

---

A Queue interface can be implemented in java by using either any one of four classes: LinkedList class, AbstractQueue class, PriorityQueue class, and ArrayDeque class. For simple FIFO queues, LinkedList and ArrayDeque classes are the best choices to create a queue.

When you need to create a queue, simply initialize a Queue variable with a LinkedList object as given below:

```
Queue<String> q = new LinkedList<>();



Queue<Integer> q = new ArrayDeque<>();
```

## Methods of Java Queue Interface

In addition to methods that Queue inherits from Collection, Queue defines several of its own methods that are as follows:

**1. boolean add(E e):** This method is used to insert the specified element into the queue if the space is available without violating capacity restrictions.
On successfully added, it returns true. It throws an IllegalStateException if no space is currently available.

A ClassCastException is thrown when an object is not compatible with elements in the queue.

A NullPointerException is thrown when we attempt to store a null object because null elements are not allowed in the queue.

IllegealArgumentException is thrown when an invalid argument is used.

IllegalStateException is thrown when we try to add an element to a fixed-length queue that is full.

**2. E element():** It is used to retrieve the element at the head of queue, but the element is not removed from the head of the queue. It throws an exception named NoSuchElementException if the queue is empty.

**3. boolean offer(E e):** It is used to insert the specified element e to the queue. This method returns true if e was added and false otherwise.
**4. E peek():** The peek() method is used to retrieve the element at the head of queue, but the element is not removed. It returns null if the queue is empty.
**5. E poll():** The poll() method is used to retrieve the element at the head of queue and removes the element in the process. This method returns null if this queue is empty.
**6. E remove():** It is used to retrieve and remove the element at head of the queue. It throws an exception named NoSuchElementException if the queue is empty.
**Methods inherited from Collection interface:**
addAll, remove, removeAll, isEmpty, clear, contains, containsAll, equals, hashCode, iterator,  retainAll, size, toArray.

---

**Note:**
1. The poll() and remove() methods are similar, except that poll() returns null object if the queue is empty, whereas remove() throws an exception named NoSuchElementException.

2. The peek() and element() methods are similar, except that peek() returns null object if the queue is empty, whereas element() throws an exception named NoSuchElementException.

3. The offer() method is used to insert an element to the queue. This method is similar to the add() method inherited from the Collection interface, but the offer() method is more preferred for queues.

## Java Queue Example Programs

---

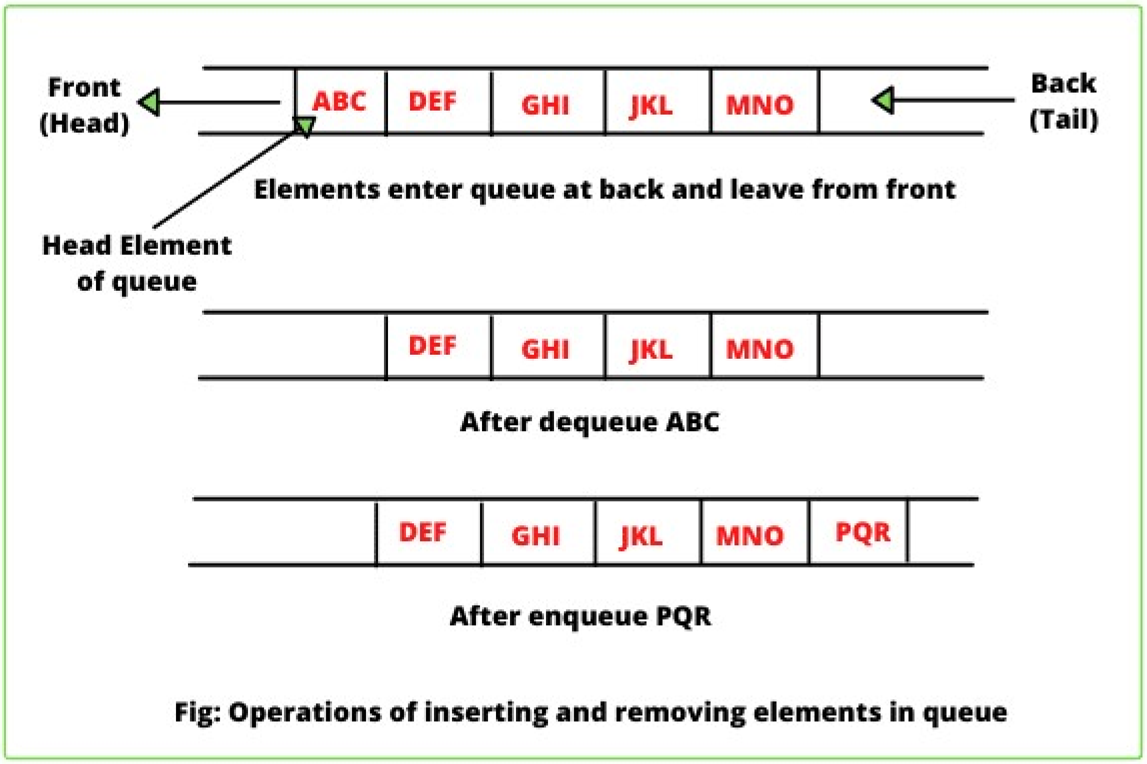
# Java Map Interface Implementation Classes

Let’s take example programs to perform various operations based on the above methods defined by queue interface.

**Program source code 1:**

```
import java.util.LinkedList;

import java.util.Queue;

public class QueueEx {

public static void main(String[] args)

{

// Create a Queue.

  Queue<String> q = new LinkedList<>();



// Adds elements to the tail of queue.

   q.add("ABC");

   q.add("DEF");

   q.add("GHI");

   q.add("JKL");

   q.add("MNO");



System.out.println("Elements in queue: " +q);

System.out.println("Head element of queue: " +q.element());

System.out.println("Removed element: " +q.remove());
System.out.println("Elements in queue after removed: " +q);



boolean addElement = q.offer("PQR");

System.out.println("Is new element added to the tail of queue: " +addElement);

System.out.println("Elements in queue after adding new element: " +q);

 }

}

Output:

     Elements in queue: [ABC, DEF, GHI, JKL, MNO]

     Head element of queue: ABC

     Removed element: ABC

     Elements in queue after removed: [DEF, GHI, JKL, MNO]

     Is new element added to the tail of queue: true

     Elements in queue after adding new element: [DEF, GHI, JKL, MNO, PQR]
```

**Explanation:**



As you can see in the above diagram, the adding operation takes place at one end of the queue called tail. This operation is known as enqueue. The removal operation takes place at the other end called head. This operation is known as dequeue.

**Program source code 2:**

```
import java.util.LinkedList;

import java.util.Queue;

public class QueueEx2 {

public static void main(String[] args)

{

// Create a Queue.

  Queue<Integer> q = new LinkedList<>();
```

```
// Check queue is empty or not.

   boolean isEmpty = q.isEmpty();

   System.out.println("Is queue empty: " +isEmpty);



   System.out.println("q.peak(): " +q.peek()); // Returns null because queue is empty.

   System.out.println("q.poll(): " +q.poll()); // Returns null because queue is empty.



// Adds elements to the tail of queue.

   q.add(10);

   q.add(20);

   q.add(30);

   q.add(25);

   q.add(50);



System.out.println("Size of queue: " +q.size());

System.out.println("Original elements in queue: " +q);



int head = q.remove();
```

```
System.out.println("Removed element at the head of queue: " +head); // Removes the
head of queue.

System.out.println("Elements in queue: " +q);



int peek = q.peek();

System.out.println("Head element of queue: " +peek); // Retrieves the head of queue
without removing.

System.out.println("Elements in queue: " +q);

 }

}

Output:

     Is queue empty: true

     q.peak(): null

     q.poll(): null

     Size of queue: 5

     Original elements in queue: [10, 20, 30, 25, 50]

     Removed element from the head of queue: 10

     Elements in queue: [20, 30, 25, 50]

     Head element of queue: 20

     Elements in queue: [20, 30, 25, 50]
```

**Program source code 3:**

## Java Map Interface Implementation Classes

```
import java.util.ArrayDeque;

import java.util.Queue;

public class QueueEx3 {

public static void main(String[] args)

{

// Create a Queue.

  Queue<Integer> q = new ArrayDeque<>();



   q.offer(50);

   q.offer(50);

   q.offer(60);

   q.offer(20);

   q.offer(10);



System.out.println(q);

System.out.println("q.element(): " + q.element());



System.out.println("q.remove(): " + q.remove());

System.out.println(q);
System.out.println("q.remove(): " + q.remove());

System.out.println(q);



System.out.println("q.offer(100): ");

  q.offer(100);

System.out.println(q);



System.out.println("q.remove(): " + q.remove());

System.out.println(q);

 }

}

Output:

      [50, 50, 60, 20, 10]

      q.element(): 50

      q.remove(): 50

      [50, 60, 20, 10]

      q.remove(): 50

      [60, 20, 10]

      q.offer(100):

      [60, 20, 10, 100]
```

```
q.remove(): 60

[20, 10, 100]
```

## PriorityQueue in Java | Methods, Example

A **PriorityQueue in Java** is a queue or collection of elements in which elements are stored in order of their priority.
It is an abstract data type similar to an ordinary queue except for its removal algorithm.

An ordinary queue is a first-in-first-out (FIFO) data structure. In FIFO, elements are added to the end of the queue and removed from the beginning.

But, In Java PriorityQueue, elements are stored in order of their priority. When accessing elements, the element with the highest priority is removed first before the element with lower priority.

PriorityQueue was added in Java 1.5 version and it is part of the Java Collection Framework. It is present in java.util.PriorityQueue package. Let's understand it with realtime examples.

### Realtime Examples of PriorityQueue in Java

1. A typical example of a priority queue is work schedule. Works are added in random order but each work has a priority. Urgent work is assigned the highest priority and is done first.

2. In a hospital, the emergency room assigns priority numbers to patients. The patient with the highest priority is checked up first.

Similarly, Java PriorityQueue class provides an implementation of a priority queue. We can create a queue with its own priority queue in java.

# Java Map Interface Implementation Classes

Priority is decided by Comparator provided in the constructor PriorityQueue(initialCapacity, Comparator) when the priority queue is instantiated.
If no comparator is provided, PrirotyQueue orders its elements according to their natural ordering using the Comparable interface. In simple words, by default, the priority is based on the natural order of the elements.

For example, if all elements are the type of Integer and no comparator is provided at the time of construction, natural ordering of elements is used to prioritize them.

The element at the head of the priority queue is the smallest element with respect to specified ordering. That is, the element having the smallest value will be assigned with the highest priority and removed first from the queue.

Look at the below figure where a new element is inserted in the queue and PriortyQueue orders its elements according to the natural order.

The element having the greatest value will be assigned with the lowest priority and it will be placed at the tail of priority queue.

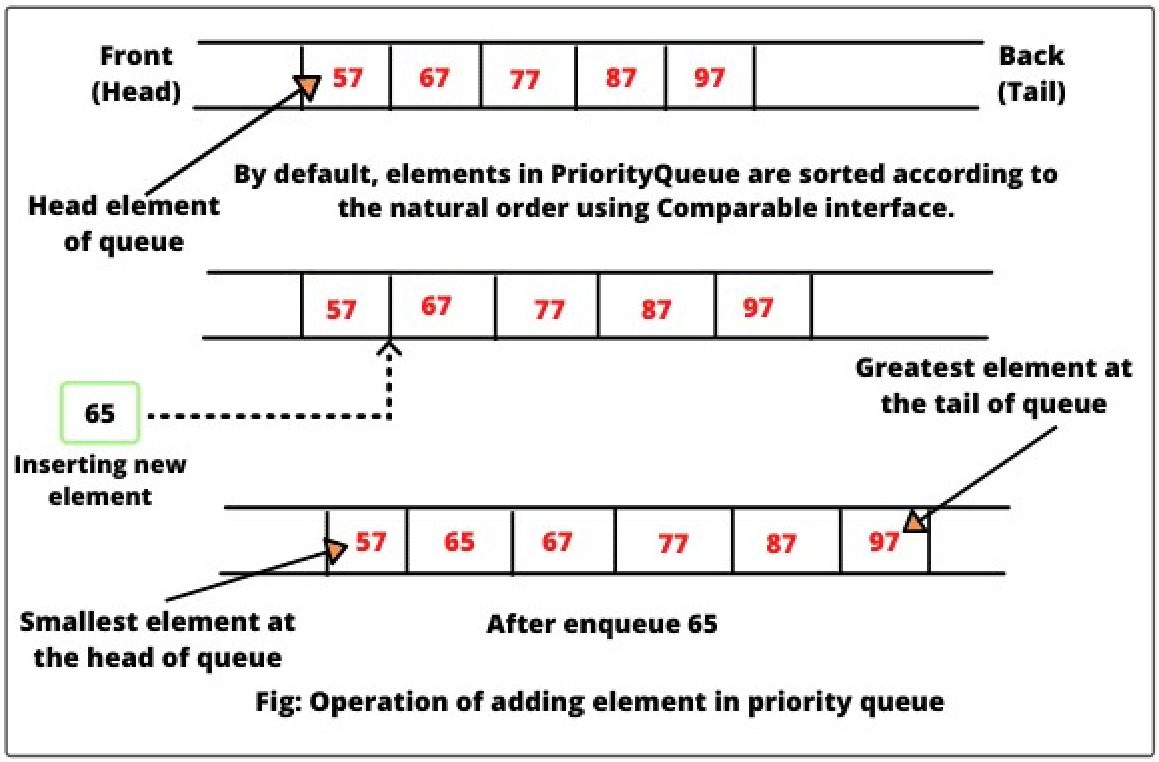

Front
(Head)
57
67
77
87
97
Back
(Tail)
Head element of queue
By default, elements in PriorityQueue are sorted according to the natural order using Comparable interface.
57
67
77
87
97
65
Inserting new element
Greatest element at the tail of queue
57
65
67
77
87
97
Smallest element at the head of queue
After enqueue 65

Fig: Operation of adding element in priority queue

Removal of elements takes place from the front end of the queue. Look at the below figure to understand better.

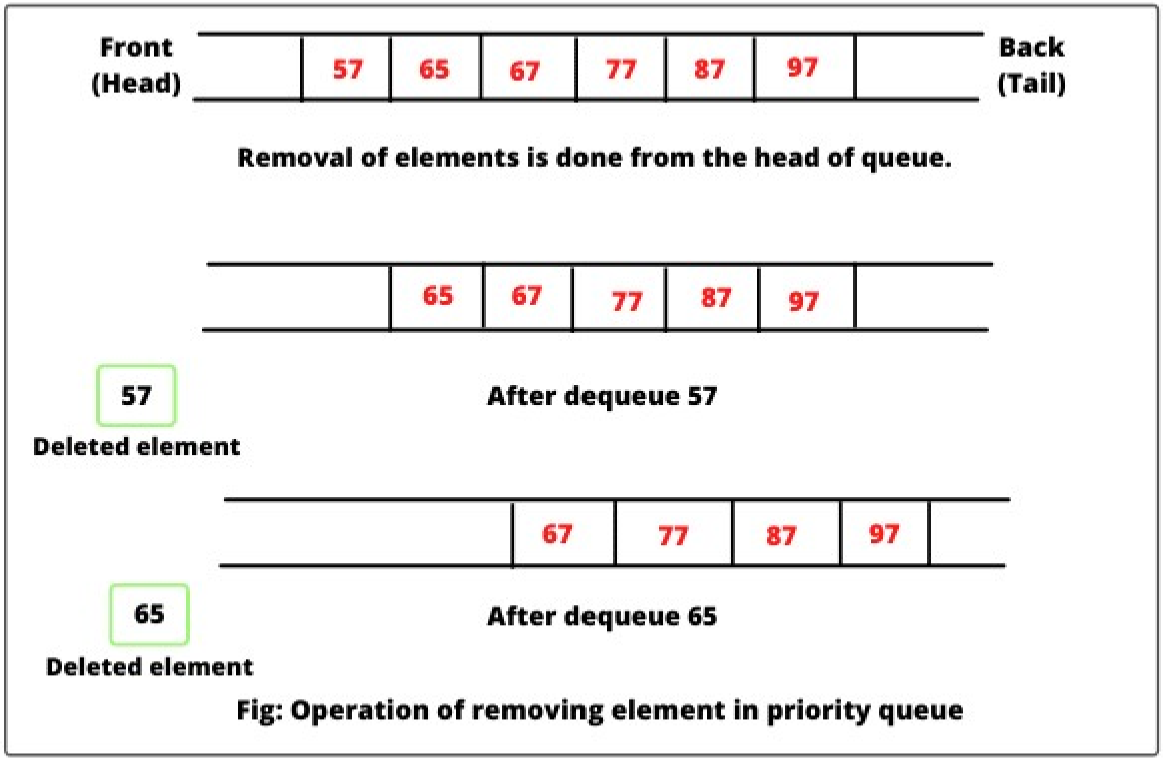


Fig: Operation of removing element in priority queue

If multiple elements are tied for the highest priority, one of those elements is arbitrarily chosen as the least element. Similarly, when there is a tie among elements at the tail of the priority queue, it is also arbitrarily chosen.

## Hierarchy of Java PriorityQueue

PriorityQueue extends AbstractQueue class to support a priority-based queue. It implements the Queue interface, Serializable interface but not implements Cloneable.
The hierarchy diagram of PriorityQueue in java is shown in the below figure.

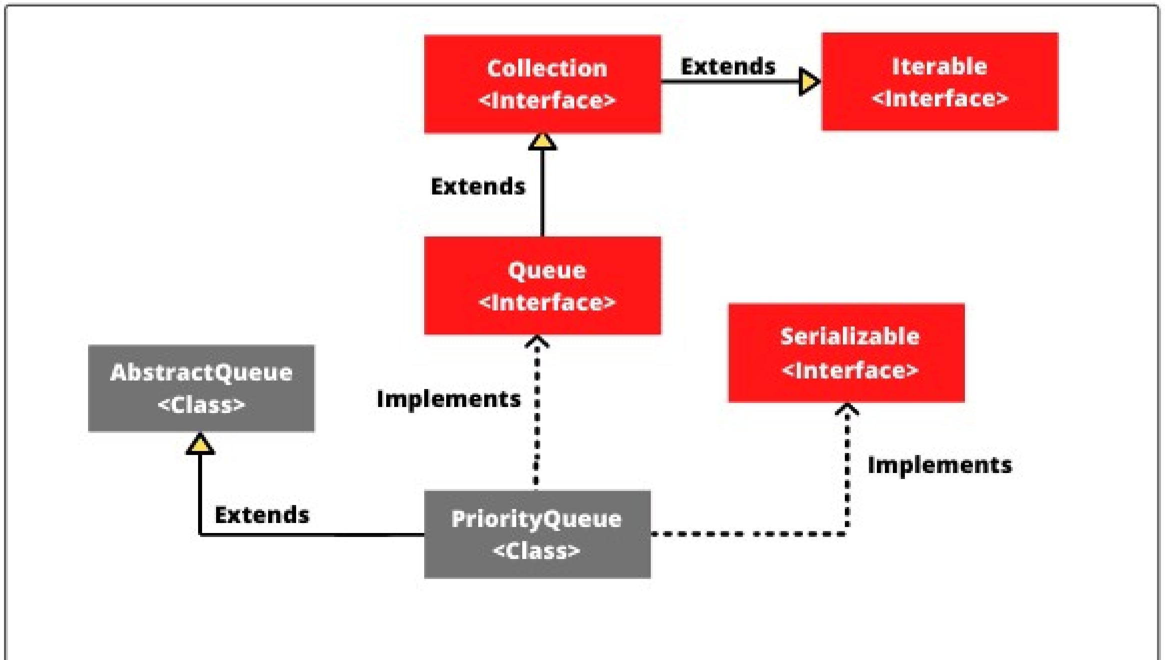


Fig: Hierarchy diagram of PriorityQueue in Java

## PriorityQueue class declaration

PriorityQueue is a generic class that can be declared as follows:

```
public class PriorityQueue<E>

   extends AbstractQueue<E>

      implements Serializable
```

Here, E represents the type of objects stored in the queue.

## Features of PriorityQueue

---

There are few important features of Priority Queue that are as follows:

1. The underlying data structure for implementing PriorityQueue in Java is Binary Heap.

2. Java PriorityQueue doesn’t permit null elements.

3. It does not allow the insertion of non-comparable objects when relying on natural ordering.

4. PriorityQueue is an implementation class for unbounded priority queue but it has an internal capacity that governs the size of an array used to store priority queue elements.

The capacity value is always at least as large as the queue size. As elements are inserted into the priority queue, its internal capacity grows automatically.

5. Element at the head of priority queue is least element.

6. Element at the tail of priority queue is greatest element.

7. Removal of elements always takes place from the front end (head) of queue.

8. PriorityQueue is not synchronized. That means it is not thread-safe. For working in a multithreading environment, Java provides a PriorityBlockingQueue class that implements the BlockingQueue interface.

9. It provides O(log(n)) time for enqueuing and dequeuing operations.

10. The iterator returned by the iterator() method does not guarantee that it will traverse elements in their sorted order.

## Constructors of PriorityQueue in Java

PriorityQueue defines six constructors that are as follows:

**1. PriorityQueue():** This form of constructor is used to create an empty, default priority queue with an initial capacity of 11 elements. It orders its elements according to their natural ordering.
The general syntax to create a PriorityQueue instance with initial capacity of 11 is as follows:

```
PriorityQueue<E> pq = new PriorityQueue<E>();
```

**2. PriorityQueue(int initialCapacity):** This constructor is used to create a default priority queue with the specified initial capacity that orders its elements according to their natural ordering.
This constructor throws an exception named IllegalArgumentException when initialCapacity is less than 1.

**3. PriorityQueue(Collection<? extends E> c):** This constructor is used to create a priority queue with the specified collection c.
It throws ClassCastException when collection c's elements cannot be compared with one another based on the priority queue's ordering.
When c or any of its elements contain null object, this constructor throws NullPointerException.

**4. PriorityQueue(int initialCapacity, Comparator<? super E> comparator):** This constructor creates a priority queue with the specified initial capacity that orders its elements according to the specified comparator.
When comparator contains null reference, natural ordering is used to sort elements. This form of constructor also throws IllegalArgumentException when initialCapacity is less than 1.

**5. PriorityQueue(PriorityQueue<? extends E> pq):** This constructor creates a priority queue containing the elements in the specified priority queue pq.
The elements in this priority queue will be ordered according to the same ordering as pq. It also throws ClassCastException when pq's elements cannot be compared with one another based on the pq's ordering.

When pq or any of its elements contain null object, this constructor throws NullPointerException.

**6. PriorityQueue(SortedSet<? extends E> ss):** This constructor creates a priority queue containing the specified sorted set ss's elements. The elements in this priority queue will be ordered according to same ordering as ss.
When ss's elements cannot be compared with one another based on the ss's ordering, this constructor throws ClassCastException. When ss or any of its elements contain null object, this constructor throws NullPointerException.

## Methods of PriorityQueue in Java

---

PriorityQueue provides the following methods. They are as follows:

**1. boolean add(E e):** This method is used to add the specified element into the priority queue.

**2. void clear():** This method is used to remove all the elements from the priority queue.
**3. comparator():** It returns the comparator used to order the elements of queue. If the queue is sorted according to the natural ordering of its elements, it returns null object.
**4. boolean contains(Object o):** It returns true if the queue contains the specified element.
**5. Iterator<E> iterator():** It returns an iterator over the elements in the priority queue.

---

**6. boolean offer(E e):** It is used to add the specified element into the priority queue.
**7. E peek():** This method retrieves, but does not remove, element at the head of the queue. It returns null element if the queue is empty.
**8. E poll():** This method retrieves and removes element at the head of the queue. It returns null if this queue is empty.
**9. boolean remove(Object o):** It is used to remove the specified element from the queue.
**10. int size():** It returns the number of elements in the queue.

---

**11. Object[] toArray():** It returns an array containing all the elements present in the queue.
**12. <T> T[ ] toArray(T[ ] a):** This method returns an array containing all the elements present in the queue; the runtime type of the returned array is that of the specified array.
**Methods inherited from AbstractQueue class:** addAll, remove. element
**Methods inherited from AbstractCollection class:** isEmpty, removeAll, retainAll, containsAll, toString
**Methods inherited from Object class:** equals, finalize, clone, getClass, hashCode, notify, notifyAll, wait,

## Java Map Interface Implementation Classes

**Methods inherited from Collection interface:** equals, hashCode, isEmpty, removeAll, retainAll, containsAll,

## Java PriorityQueue Example Programs

---

Let's take some example programs to perform various operations based on the above methods provided by Java PriorityQueue.

**Program source code 1:**

```
import java.util.Iterator;

import java.util.PriorityQueue;

public class PriorityQueueEx {

public static void main(String[] args)

{

// Create a Queue. This priority queue stores Strings objects.

  PriorityQueue<String> pq = new PriorityQueue<>();



// Adds elements to the priority queue.

  pq.offer("USA");

  pq.offer("India");

  pq.offer("England");

  pq.offer("Germany");

  pq.offer("Australia");
System.out.println("Priority queue: " +pq);



// Iterating elements of priority queue.

  System.out.println("Iterating elements of priority queue");

  Iterator<String> iterator = pq.iterator();

  while (iterator.hasNext()) {

   System.out.print(iterator.next() + " ");

  }

 }

}

Output:

    Priority queue: [Australia, England, India, USA, Germany]

    Iterating elements of priority queue

    Australia England India USA Germany
```

**Explanation:** As you will notice in the output of the above program when you print the queue, its elements are not ordered according to their priority.
This is because a queue is never used to iterate over its elements. PriorityQueue class does not guarantee any ordering of elements when we use an iterator. Therefore, when we print the priority queue, its elements are not ordered according to their priority.

However, when we will use peek() or remove() method, the correct element will be peeked at or removed, which is based on the element's priority.

---

Let's create a program where we will peek or remove the correct element based on priority.

**Program source code 2:**

```
import java.util.PriorityQueue;

public class PriorityQueueEx2 {

public static void main(String[] args)

{

PriorityQueue<String> pq = new PriorityQueue<>();



  pq.offer("USA");

  pq.offer("India");

  pq.offer("England");

  pq.offer("Germany");

  pq.offer("Australia");



System.out.println("Elements in queue: " +pq);
while (pq.peek() != null) {

  System.out.println("Head Element: " + pq.peek());

  System.out.println("Removed Element from Queue: " +pq.remove());

  System.out.println("Priority queue: " + pq);

  }

 }

}

Output:

    Elements in queue: [Australia, England, India, USA, Germany]

    Head Element: Australia

    Removed Element from Queue: Australia

    Priority queue: [England, Germany, India, USA]

    Head Element: England

    Removed Element from Queue: England

    Priority queue: [Germany, USA, India]

    Head Element: Germany

    Removed Element from Queue: Germany

    Priority queue: [India, USA]

    Head Element: India

    Removed Element from Queue: India
    Priority queue: [USA]

    Head Element: USA

    Removed Element from Queue: USA

    Priority queue: []
```

**Explanation:** As you can observe in the output, when we use the peek() and remove() methods, the correct element is peeked at and removed from the queue, which is based on the element’s priority.
This is because PriorityQueue removes one element from it, processes that element, and then removes another element.

**Program source code 3:**

```
import java.util.Collections;

import java.util.PriorityQueue;

public class PriorityQueueEx {

public static void main(String[] args)

{

// Create a Queue. This priority queue stores Integer objects.

   PriorityQueue<Integer> pq = new PriorityQueue<>();



// Adds elements to the priority queue.

  pq.offer(50);

  pq.offer(100);

  pq.offer(60);
```

```
   pq.offer(20);

   pq.offer(10);



System.out.println("Priority queue using Comparable:");

while(pq.size() > 0) {

   System.out.print(pq.remove() + " ");

}

PriorityQueue<Integer> pq2 = new PriorityQueue<>(5, Collections.reverseOrder());

  pq2.offer(50);

  pq2.offer(100);

  pq2.offer(60);

  pq2.offer(20);

  pq2.offer(10);



int size = pq2.size();

System.out.println("\nSize of priority queue: " +size);

System.out.println("\nPriority queue using Comparator:");

while(size > 0) {

    System.out.print(pq2.remove() + " ");

   }
```

```
 }

}

Output:

      Priority queue using Comparable:

      10 20 50 60 100

      Size of priority queue: 5



      Priority queue using Comparator:

      100 60 50 20 10
```

## Deque in Java | Methods, Example

A **double ended queue** or **deque in Java** is a linear data structure that supports insertion and removal of elements from both ends (head and tail). It is usually pronounced "deck", not "de queue".
Java deque is an extended version of queue to handle a double-ended queue. It extends Queue interface with additional methods for adding and removing elements from both ends of queues.

Deque interface was recently introduced in Java 1.6 version. It is now part of the Java collections framework. It is present in java.util.Deque package. Double ended queue in Java can be used as FIFO (first-in, first-out) queue or LIFO (last-in, first-out) queue. Adding elements in the middle of queue is not supported.

### Hierarchy of Java Deque Interface

# Java Map Interface Implementation Classes

Java Deque interface extends Queue interface to handle double ended queue. It is implemented by LinkedList and ArrayDeque classes, both of which can be used to create a queue whose size can grow as needed. Java LinkedList is ideal for queue operations because it is efficient for adding and removing elements from both ends of a list.

Other implementation classes are ConcurrentLinkedDeque, and LinkedBlockingDeque which also implement the deque interface.

The hierarchy diagram of deque interface in java is shown in the below figure.

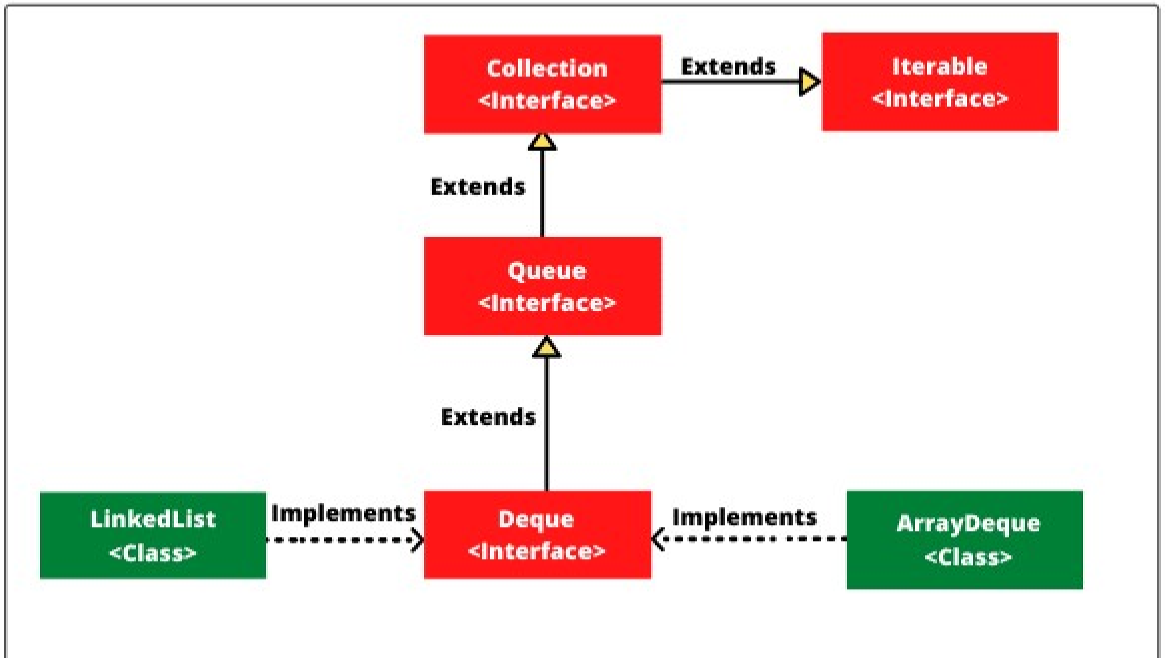


Fig: Hierarchy diagram of Deque in Java

## Deque Interface declaration

Deque is a generic interface that can be declared as follows:

```
public interface Deque<E>
```

```
    extends Queue<E>
```

Here, E represents the type of objects that deque will hold.

## Features of Java Double ended queue

There are several interesting features of the deque in java that is as follows:

1. Java Deque interface orders elements in First In First Out or Last In First Out policy.

2. Deque does not allow to store null elements. If we try to add, NullPointerException will be thrown.

3. The most important features of Deque are push and pop. The push() and pop() methods are commonly used to enable a Deque to function as a stack.

To put an element on the top of the stack, call push( ) method. To remove the top element, call pop( ) method.

4. Elements can be retrieved and removed from both ends of the queue.

5. Elements can be added from both ends of the queue.

6. The LinkedList and ArrayDeque classes are two implementation classes for the Deque interface.

7. ArrayDeque class can be implemented if deque is used as a LIFO queue (or a stack).

8. The LinkedList implementation performs better if deque is used as a FIFO queue (or simply as a queue).

## Deque Methods in Java

# Java Map Interface Implementation Classes

In addition to methods inheriting from the Collection interface and Queue, Deque interface adds 17 new methods to facilitate insertion, removal, and peeking operations at both ends. They are as follows:

**1. void addFirst(E e):** This method is used to add an element at the head of the deque. It throws an exception named IllegalStateException if a capacity-restricted deque is out of space.
**2. void addLast(E e):** This method is used to add an element at the tail of the deque. It throws an IllegalStateException if a capacity-restricted deque is out of space.
**3. boolean offerFirst(E e):** It is used to add an element at the head of the deque. It returns true if the element is added successfully in the deque otherwise returns false. However, it does not throw an exception on failure.
**4. boolean offerLast(E e):** It is used to add an element at the tail of the deque. It returns true if the element is added successfully in the deque otherwise returns false. However, it does not throw an exception on failure.
**5. E removeFirst( ):** This method is used to retrieve and remove the element from the head of the deque. They throw an exception named NoSuchElementException if the deque is empty.

---

**6. E removeLast():** This method is used to retrieve and remove the element from the tail of the deque. They throw an exception named NoSuchElementException if the deque is empty.
**7. E pollFirst( ):** The pollFirst() method performs the same task as the removeFirst() method. However, it returns null if the deque is empty.
**8. E pollLast():** The pollLast() method performs the same task as the removeLast() method. However, it returns null if the deque is empty.

**9. E getFirst( ):** It is used to retrieve without removing the first element from the head of deque. It throws NoSuchElementException if the deque is empty.
**10. E getLast():** It is used to retrieve without removing the last element from the tail of deque. It throws NoSuchElementException if the deque is empty.

---

**11. E peekFirst( ):** The peekFirst() method works the same task as the getFirst() method. It returns null object if deque is empty instead of throwing an exception.
**12. E peekLast():** The peekLast() method works the same task as the getLast() method. It returns null object if deque is empty instead of throwing an exception.
**13. void push(E e ):** The push() method adds (or pushes) an element to the head of the deque. It will throw an IllegalStateException if a capacity-restricted deque is out of space. This method is the same as addFirst() method.
**14. E pop( ):** The pop() method removes (or pops) an element from the head of deque. If the deque is empty, it will throw NoSuchElementException. This method is the same as removeFirst() method.
**15. boolean removeFirstOccurrence(Object o):** This method removes the first occurrence of the specified element from the deque. It returns true if successful removed and false if the deque did not contain the specified element.

**16. boolean removeLastOccurrence(Object o):** This method removes the last occurrence of the specified element from the deque. It will return true if successful removed and false if the deque did not contain the specified element.

**17. Iterator<E> descendingIterator( ):** It returns an iterator object that iterate over its elements in reverse order (from tail to head). The descendingIterator() method works the same as the iterator() method, except that it moves in the opposite direction.

## Java Deque Implementation Example Programs

---

Let's take an example program where we will perform operations based on getFirst(), removeFirst(), addFirst(), getLast(), removeLast(), and addLast() methods. We will implement ArrayDeque class for deque interface.

**Program source code 1:**

```
import java.util.ArrayDeque;

import java.util.Deque;



public class DequeTestEx {

public static void main(String[] args)

{

// Create a Deque and add elements to the deque.

  Deque<String> dq = new ArrayDeque<String>();



  dq.offer("ABC");

  dq.offer("PQR");

  dq.offer("MNO");
  dq.offer("IJK");

  dq.offer("GHI");



System.out.println(dq);



System.out.println("dq.getFirst(): " +dq.getFirst());

System.out.println(dq);



System.out.println("dq.removeFirst(): " +dq.removeFirst());

System.out.println(dq);



// Adding new element at the head of queue.

   dq.addFirst("ABC");

   System.out.println(dq);



System.out.println("dq.getLast(): " +dq.getLast());

System.out.println(dq);



System.out.println("dq.removeLast(): " +dq.removeLast());

System.out.println(dq);
```

```
dq.addLast("GHI");

System.out.println(dq);

 }

}

Output:

     [ABC, PQR, MNO, IJK, GHI]

     dq.getFirst(): ABC

     [ABC, PQR, MNO, IJK, GHI]

     dq.removeFirst(): ABC

     [PQR, MNO, IJK, GHI]

     [ABC, PQR, MNO, IJK, GHI]

     dq.getLast(): GHI

     [ABC, PQR, MNO, IJK, GHI]

     dq.removeLast(): GHI

     [ABC, PQR, MNO, IJK]

     [ABC, PQR, MNO, IJK, GHI]
```

## Deque Implementation in Java using LinkedList

Let's create a program where we will use a deque as a FIFO queue (or simply as a queue). We will implement LinkedList class for the deque interface. LinkedList class performs better if we use deque as a FIFO queue. Look at the source code to understand better.

**Program source code 2:** Using a deque as a FIFO queue

```
import java.util.Deque;

import java.util.LinkedList;

import java.util.NoSuchElementException;



public class DequeAsQueue {

public static void main(String[] args)

{

// Create a Deque and add elements at its tail using addLast() or offerLast() method

  Deque<String> dq = new LinkedList<>();



   dq.addLast("John");

   dq.offerLast("Richard");

   dq.offerLast("Donna");

   dq.offerLast("Ken");

   dq.offer("Peter");



System.out.println("Deque: " + dq);
```

```
// Remove elements from deque until it is empty.

 while (dq.peekFirst() != null)

{

   System.out.println("Head Element: " + dq.peekFirst());

   System.out.println("Removed element from Deque: " +dq.removeFirst());

   System.out.println("Elements in Deque: " + dq);

 }

System.out.println("\n");

// Now, deque is empty. Try to call its peekFirst(), getFirst(), pollFirst() and
removeFirst() methods.



 System.out.println("deque.isEmpty(): " + dq.isEmpty());

 System.out.println("deque.peekFirst(): " + dq.peekFirst());

 System.out.println("deque.pollFirst(): " + dq.pollFirst());



try {

   String str = dq.getFirst();

   System.out.println("deque.getFirst(): " + str);

}
```

```
catch (NoSuchElementException e) {

   System.out.println("deque.getFirst(): Deque is empty.");

}

try {

   String str = dq.removeFirst();

   System.out.println("deque.removeFirst(): " + str);

}

catch (NoSuchElementException e) {

   System.out.println("deque.removeFirst(): Deque is empty.");

  }

 }

}

Output:

      Deque: [John, Richard, Donna, Ken, Peter]

      Head Element: John

      Removed element from Deque: John

      Elements in Deque: [Richard, Donna, Ken, Peter]

      Head Element: Richard

      Removed element from Deque: Richard

      Elements in Deque: [Donna, Ken, Peter]
```

```
Head Element: Donna

Removed element from Deque: Donna

Elements in Deque: [Ken, Peter]

Head Element: Ken

Removed element from Deque: Ken

Elements in Deque: [Peter]

Head Element: Peter

Removed element from Deque: Peter

Elements in Deque: []



deque.isEmpty(): true

deque.peekFirst(): null

deque.pollFirst(): null

deque.getFirst(): Deque is empty.

deque.removeFirst(): Deque is empty.
```

## Deque Implementation in Java using ArrayDeque

---

Let’s create another program where we will demonstrate how to use a Deque as a stack (or LIFO queue). We will implement ArrayDeque class for deque interface.

**Program source code 3:** Using Deque as Stack (or LIFO queue)

```
import java.util.ArrayDeque;

import java.util.Deque;

public class DequeAsStack {

public static void main(String[] args)

{

// Create a Deque and use it as stack. add elements at its tail using addLast() or
offerLast() method.

  Deque<String> dq = new ArrayDeque<>();



  dq.push("John");

  dq.push("Richard");

  dq.push("Donna");

  dq.push("Ken");

  dq.push("Peter");



System.out.println("Stack: " + dq);



// Remove all elements from the deque.

 while (dq.peek() != null)

{
  System.out.println("Element at top: " + dq.peek());

  System.out.println("Popped: " + dq.pop());

  System.out.println("Stack: " + dq);

 }

 System.out.println(" Is Stack empty: " + dq.isEmpty());

 }

}

Output:

     Stack: [Peter, Ken, Donna, Richard, John]

     Element at top: Peter

     Popped: Peter

     Stack: [Ken, Donna, Richard, John]

     Element at top: Ken

     Popped: Ken

     Stack: [Donna, Richard, John]

     Element at top: Donna

     Popped: Donna

     Stack: [Richard, John]

     Element at top: Richard

     Popped: Richard
```

```
Stack: [John]

Element at top: John

Popped: John

Stack: []

Is Stack empty: true
```

## How to Iterate over Deque in Java?

---

Let’s take an example program where we will iterate over elements of deque using iterator() method. The iterator() method returns an iterator object that iterates over elements of a deque in LIFO order. i.e. the elements will be traversed from head to tail.

**Program source code 4:**

```
import java.util.ArrayDeque;

import java.util.Deque;

import java.util.Iterator;



public class IteratingDeque {

public static void main(String[] args)

{

 Deque<Integer> dq = new ArrayDeque<Integer>();
```

```
  dq.offer(50);

  dq.offer(10);

  dq.offer(20);

  dq.offer(05);

  dq.offer(30);


System.out.println("Elements in deque:");

System.out.println(dq);


// Iterating over elements of the deque.

System.out.println("\nIteration in forward direction:");

 Iterator<Integer> itr = dq.iterator();

 while(itr.hasNext())

{

   System.out.println(itr.next());

 }

// Iterating over elements in reverse order.

 System.out.println("\nIteration in reverse order:");

 Iterator<Integer> itr2 = dq.descendingIterator();

 while(itr2.hasNext())
 {
    System.out.println(itr2.next());
  }
 }
}
Output:
     Elements in deque:
    [50, 10, 20, 5, 30]

    Iteration in forward direction:
    50
    10
    20
    5
    30

    Iteration in reverse order:
    30
    5
    20
```

```
   10

   50
```

Let’s take another example program where we will use enhanced for loop to iterate over elements of deque in LIFO order. Look at the source code.

**Program source code 5:**

```
import java.util.ArrayDeque;

import java.util.Deque;

public class DequeAsQueue {

public static void main(String[] args)

{

 Deque<Integer> dq = new ArrayDeque<Integer>();



  dq.offer(50);

  dq.offer(10);

  dq.offer(20);

  dq.offer(05);

  dq.offer(30);



// Iterating over elements of deque using enhanced for loop.

 System.out.println("Iterating using enhanced for loop");
```

```
  for (Integer element: dq) {
      System.out.println(element);
  }
 }
}
Output:
       Iterating using enhanced for loop
       50
       10
       20
       5
       30
```

## ArrayDeque in Java | Methods, Example

**ArrayDeque in Java** is a concrete class that creates a dynamic array (resizable array) to store its elements from both sides of the queue. It provides a resizable-array implementation of deque interface. Therefore, it is also known as array double-ended queue or simply array deck.

Since ArrayDeque has no capacity restrictions, ArrayDeque is especially useful for implementing stacks and queues whose sizes are not known in advance.

## Java Map Interface Implementation Classes

ArrayDeque class was added by Java 6 version that makes it a recent addition to the Java Collections Framework. It is present in java.util.ArrayDeque package.

### Hierarchy of ArrayDeque in Java

---

Java ArrayDeque class implements double-ended queue interface to support the addition of elements from both sides of queue. It also implements queue interface to support the first in first out data structure. All implemented interfaces by ArrayDeque class in the hierarchy are Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, Queue<E>.

ArrayDeque class extends AbstractCollection. The hierarchy diagram of ArrayDeque is shown in the below figure.

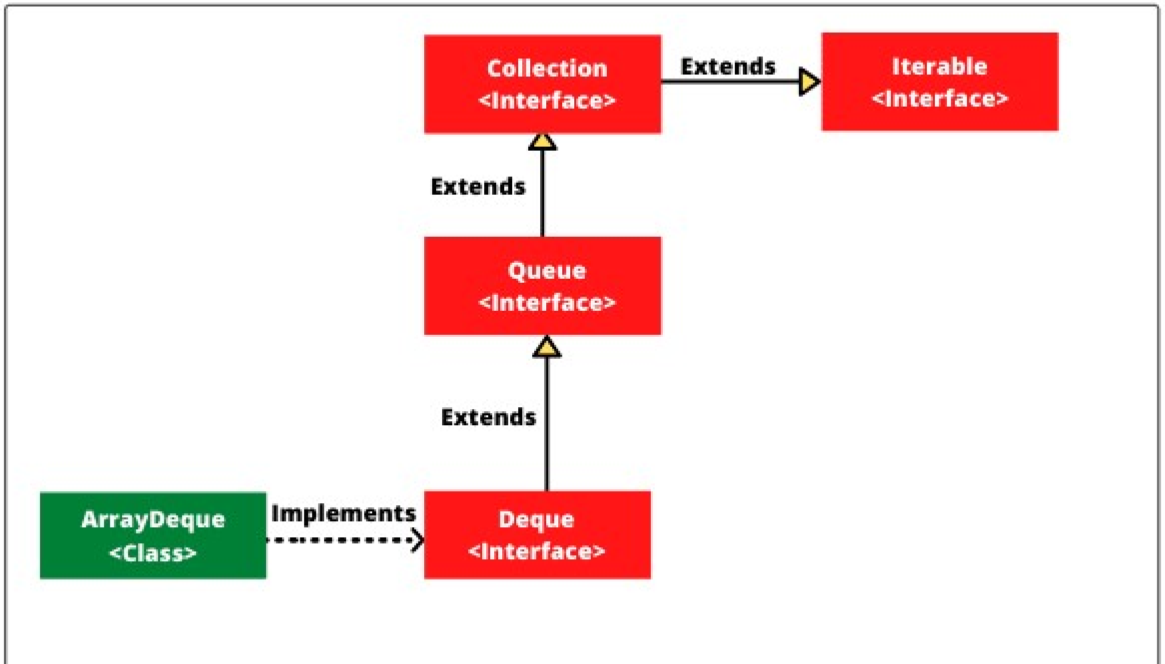


Fig: Hierarchy diagram of ArrayDeque in Java

## ArrayDeque class declaration

ArrayDeque is a generic class that can be declared as follows:

```
public class ArrayDeque<E>

  extends AbstractCollection<E>

     implements Deque<E>, Cloneable, Serializable
```

Here, E represents the type of objects stored in the collection.

## Features of ArrayDeque

There are several important features of array deck that should be kept in mind. They are as:

1. Java ArrayDeque class provides a resizable array implementation of Deque interface.
2. It has no capacity restrictions. So, it can grow according to the need to handle elements added to the collection.

3. Array deque is not synchronized. That means it is not thread-safe. Multiple threads can access the same ArrayDeque object at the same time.

4. Null elements are restricted in the ArrayDeque.

5. ArrayDeque class performs faster operations than Stack when used as a stack.

6. ArrayDeque class performs faster operations than LinkedList when used as a queue.

7. The iterators returned by array deque class are fail-fast: If the deque is modified during iteration, the iterator will generally throw an exception named ConcurrentModificationException.

## Constructors of ArrayDeque class in Java

ArrayDeque class defines the following constructors that are as follows:

**1. ArrayDeque():** This constructor creates an empty array deque with starting capacity of 16 elements. The general syntax to create array deque object is as follows:

```
ArrayDeque<E> dq = new ArrayDeque<E>();
```

**2. ArrayDeque(int numElements):** This constructor creates an empty array deque with the specified initial capacity sufficient to hold elements. If the argument passed to numElements is less than or equal to zero, no exception will be thrown.
The general syntax to create ArrayDeque object with the specified initial capacity is as follows:

```
ArrayDeque<E> dq = new ArrayDeque<E>(int numElements);



For example:

ArrayDeque<String> dq = new ArrayDeque<String>(5);
```

**3. ArrayDeque(Collection c):** This constructor creates an array deque that is initialized with elements of collection c. If c contains null reference then NullPointerException will be thrown.
The general syntax to create ArrayDeque instance with the specified collection is given below:

```
ArrayDeque<E> dq = new ArrayDeque<E>(Collection c);
```

In all cases, the capacity can grow as per the requirement to handle elements inserted into the collection.

## Methods of ArrayDeque in Java

---

ArrayDeque in Java adds no methods of its own. All methods are inherited by Deque, Queue, and Collection interface. The important methods are given below:

**1. boolean add(Object o):** It inserts the specified element to the end of deque.
**2. void addFirst(Object o):** It inserts the specified element to the front of the deque.
**3. void addLast(Object o):** It inserts the specified element to the last of deque.
**4. void clear():** This method is used to remove all elements from deque.
**5. Object clone():** It is used to get a copy of ArrayDeque instance.

---

**6. boolean contains(Object element):** It returns true if the deque contains the specified element.
**7. Object element():** This method is used to retrieve an element from the head of deque. It does not remove element.
**8. Object getFirst():** This method is used to retrieve the first element of deque. It does not remove element.
**9. Object getLast():** This method is used to retrieve the last element of deque. It does not remove element.
**10. boolean isEmpty():** The isEmpty() method returns true if deque has no elements. 11. boolean offer(Object element): It inserts the specified element at the end of deque.

**11. boolean offerFirst(Object element):** It inserts the specified element at the front of deque.

**12. boolean offerLast(Object element):** It inserts the specified element at the end of deque.

**13. Object peek():** The peek() method retrieves element from the head of deque but does not remove. If the deque is empty, it returns null element.

**14. Object peekFirst():** The peekFirst() method retrieves the first element from the head of deque but does not remove it. It returns null element if the deque is empty.

**15. Object peekLast():** The peekLast() method retrieves the last element from deque but does not remove it. It returns null element if the deque is empty.

**16. Object poll():** The poll() method retrieves and removes the element from the head of deque. It returns null element if the deque is empty.

**17. Object pollFirst():** The pollFirst() method retrieves and removes the first element of deque. It returns null element if the deque is empty.

**18. Object pollLast():** The pollLast() method retrieves and removes the last element of deque. It returns null element if the deque is empty.

**19. Object pop():** The pop() method is used to pop an element from the stack represented by the deque.

**20. Object push(Object element):** The push() method is used to push an element from the stack represented by the deque.

**21. Object remove():** The remove() method is used to retrieve and remove an element from the deque.
**22. boolean removeFirst():** This method is used to retrieve and remove the first element from the deque.
**23. boolean removeFirstOccurrence(Object o):** This method removes the last occurrence of the specified element from the deque.
**24. int size():** It returns the number of elements in the deque.
**25. Object toArray():** It returns an array that contains all elements in the deque in the proper sequence from first to last element.

---

**26. Iterator iterator():** It returns an iterator over elements from the deque.
**27. Iterator descendingIterator():** It returns an iterator over elements in reverse sequential order from the deque.

## Java ArrayDeque Example Programs

---

Let's take some useful example programs to perform the various operations based on the above methods.

**1. Adding elements:** Let's create a program where we will add elements to the ArrayDeque using methods such as add(), addFirst(), addLast(), offer(), offerFirst(), offerLast().
**Program source code 1:**

```
import java.util.ArrayDeque;

public class ArrayDequeEx {

public static void main(String[] args)

{
```

```
// Create object of ArrayDeque class of String type.

  ArrayDeque<Integer> dq = new ArrayDeque<Integer>();



// Adding elements to deque using add() method.

    dq.add(10);

    dq.addFirst(20);

    dq.addLast(05);



// Adding elements to deque using offer() method.

    dq.offer(30);

    dq.offerFirst(50);

    dq.offerLast(40);



System.out.println("Elements in ArrayDeque : " + dq);

 }

}

Output:

      Elements in ArrayDeque : [50, 20, 10, 5, 30, 40]
```

**2. Accessing Elements:** Let's create a program where we will access elements using methods like getFirst(), getLast(), etc.

**Program source code 2:**

```
import java.util.ArrayDeque;

public class ArrayDequeEx2 {

public static void main(String[] args)

{

// Creating an empty ArrayDeque instance.

   ArrayDeque<Integer> dq= new ArrayDeque<Integer>();



    dq.add(25);

    dq.add(50);

    dq.add(75);

    dq.add(100);

    dq.add(125);



// Displaying elements of ArrayDeque.

  System.out.println("ArrayDeque: " + dq);

  System.out.println("First element is: " + dq.getFirst());

  System.out.println("Last element is: " + dq.getLast());

  }

}
```

```
Output:

     ArrayDeque: [25, 50, 75, 100, 125]

     First element is: 25

     Last element is: 125
```

**3. Removing Elements:** Let's make a program where we will remove an element from a deque using various methods available such as removeFirst(), removeLast(), poll(), pop(), pollFirst(), pollLast().

**Program source code 3:**

```
import java.util.ArrayDeque;

public class ArrayDequeEx3 {

public static void main(String[] args)

{

ArrayDeque<String> dq= new ArrayDeque<String>();



  dq.add("One");

  dq.addFirst("Two");

  dq.addLast("Three");



// Displaying elements of ArrayDeque.

   System.out.println("Elements in ArrayDeque : " + dq);
// Remove element as a stack from top/front end.

   System.out.println(dq.pop());



// Remove element as a queue from front

   System.out.println(dq.poll());



// Remove element from front end.

   System.out.println(dq.pollFirst());



// Remove element from back end.

   System.out.println(dq.pollLast());

  }

}

Output:

     Elements in ArrayDeque : [Two, One, Three]

     Two

     One

     Three

     null
```

**4. Iterating over elements of Deque:** ArrayDeque can be iterated over elements from both ends of deque using iterator() and descendingIterator() methods. Look at the following source code.

**Program source code 4:**

```
import java.util.ArrayDeque;

import java.util.Iterator;

public class ArrayDequeEx4 {

public static void main(String[] args)

{

ArrayDeque<String> dq= new ArrayDeque<String>();



  dq.add("One");

  dq.addFirst("Two");

  dq.addLast("Three");

  dq.add("Four");



// Iterating over elements from the front end of the queue using iterator() method.

  System.out.println("Front End Iteration:");

  Iterator<String> itr = dq.iterator();

  while(itr.hasNext()){

              System.out.println(itr.next());
  }


System.out.println();


// Iterating over elements in reverse order.

  System.out.println("Back End Iteration:");

  Iterator<String> itr2 = dq.descendingIterator();

  while(itr2.hasNext()){

                System.out.println(itr2.next());

  }

 }

}

Output:

      Front End Iteration:

      Two

      One

      Three

      Four


      Back End Iteration:
```

```
Four

Three

One

Two
```

**5. ArrayDeque as a Stack:** Let’s take an example program where we will use ArrayDeque to create a Stack. Look at the following source code to understand better.

**Program source code 5:**

```
import java.util.ArrayDeque;

public class ArrayDequeAsStack {

public static void main(String[] args)

{

ArrayDeque<String> dq= new ArrayDeque<String>();



String[] weekdays = {"Sunday", "Monday", "Tuesday", "Thursday", "Friday",
"Saturday"};

System.out.println("Poping Stack: ");



for(String weekday: weekdays)

            dq.push(weekday);

while(dq.peek() != null){

   System.out.println(dq.pop());
}
 }
}
Output:
     Poping Stack:
     Saturday
     Friday
     Thursday
     Tuesday
     Monday
     Sunday
```

## Stack in Java | Methods, Example Program

A **Stack in Java** is a collection (or group) of elements stored in the last in first out (LIFO) order. In other words, a stack is a data structure that stores data in last-in, first-out fashion.
This means an element that is stored as a last element into the stack, will be the first element to be removed from the stack. Only the top element on the stack is accessible at a given time.

When an element (object) is inserted into the stack, it is called **push operation**. We can create a stack of any type of elements.
When an element is removed from the stack, it is called **pop operation**.
When an element is searched in the stack, it is called **operation**.
Insertion and deletion of elements take place only from one side of the stack, traditionally called the top of the stack. That is, new elements are

added to the top of the stack, and elements are removed from the top of the stack.

Therefore, stack is called last-in, first-out data structure. A typical example of stack in java is shown in the below figure.

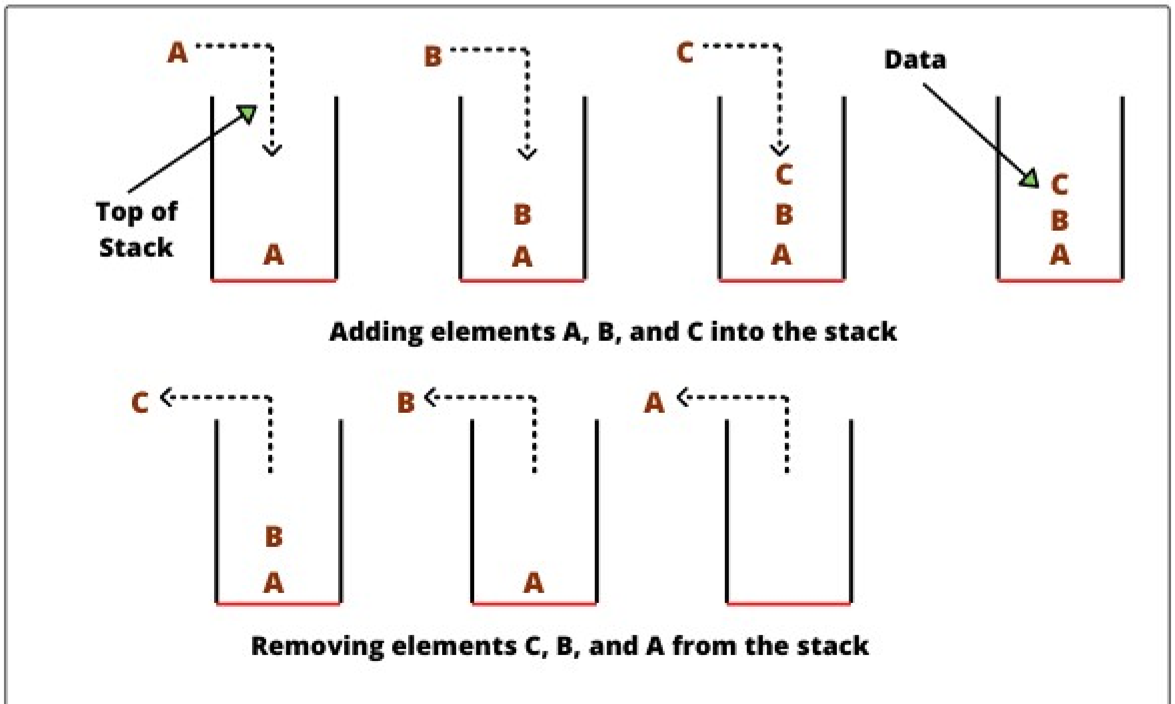

A
Top of Stack
A
B
B
A
C
C
B
A
Data
C
B
A
Adding elements A, B, and C into the stack
C
B
A
B
A
A
Removing elements C, B, and A from the stack

Fig: A stack holds elements in a Last-In, First-Out fashion

As you can see in the above figure, we have added three elements A, B, and C into the stack. When they are removed from the stack, we obtain elements C, B, and A fashion. This fashion is called **Last-In, First-Out data structure**.
Let’s understand it with realtime examples to clear more.

## Realtime Example of Stack in Java

---

1. A realtime example of stack is a stack of books as shown in the below figure. We can not take a book from the stack without removing books that are first stacked on top of it.

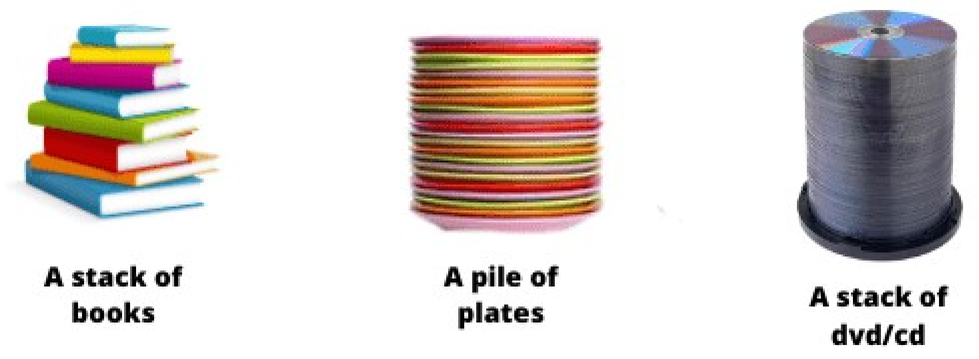


Fig: Realtime examples of Stack

2. A second real-time example of stack is a pile of plates in a cafeteria where last washed plate will be out first for use.

3. Similarly, a DVD disk holder where CDs are arranged in such that the last CD will be out first for use. This is also an example of stack.

Stack class was introduced in Java 1.0 version. It is present in java.util.Stack<E> package. This class is now considered as legacy class in java because it is not consistent with Java Collections Framework.

Java API recommends using Java ArrayDeque class at the place of stacks. ArrayDeque provides all the normal functionality of a stack and it is consistent with the JCF.

## Hierarchy of Java Stack Class

---

Java Stack class extends vector class that extends AbstractList class. It implements List interface and RandomAccess interface. Stack class also implements Serializable and Cloneable interfaces.
The hierarchy diagram of stack in java is shown in the below figure.

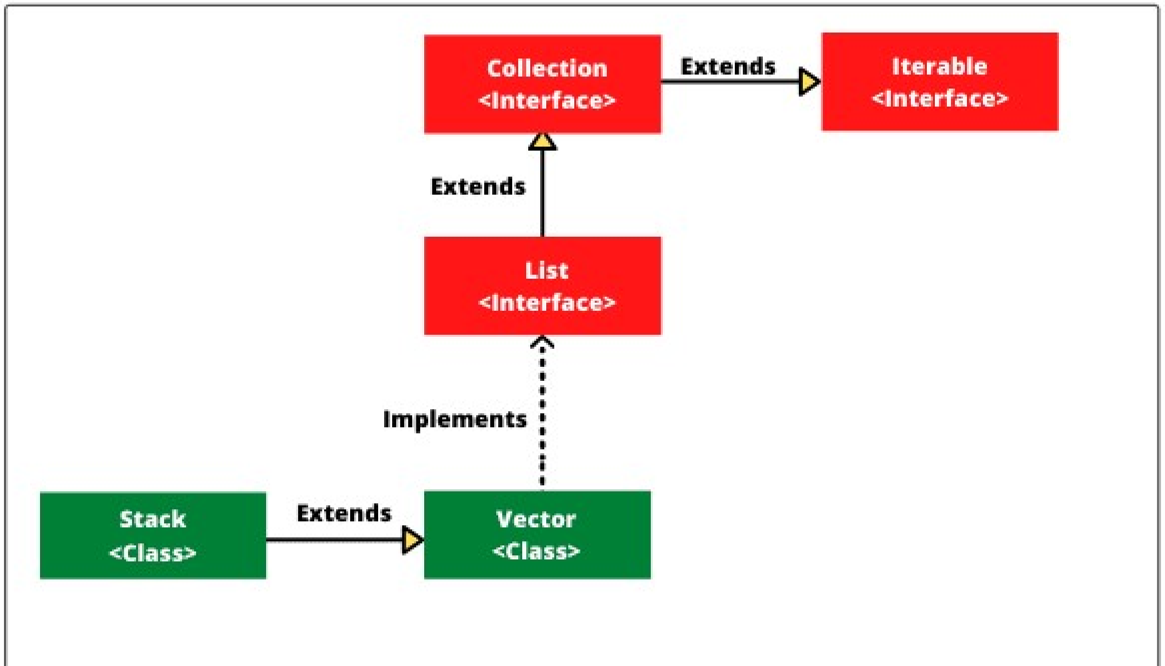


Fig: Hierarchy diagram of Stack in Java

## Stack Class declaration

---

Stack is a generic class in java that has the following declaration in a general form:

```
public class Stack<E>

      extends Vector<E>
```

Here, E represents the type of elements that stack can hold.

## Features of Stack class

---

There are several features of stack in Java that are as follows:

## Java Map Interface Implementation Classes

1. Stack is a group of elements with “last-in, first-out” retrieval.
2. In the Java stack, all operations take place at the top of the stack.
3. Push operation inserts an element to the top of stack.
4. Pop operation removes an element from the stack and returns it.
5. Stack class is synchronized. That means it is thread-safe.
6. Null elements are allowed into the stack.
7. Duplicate elements are allowed into the stack.

### Constructor of Stack Class in Java

Stack class has provided only one constructor that is as follows:

**1. Stack():** This constructor is used to create an empty stack object. The general form to create a stack object is as follows:

```
Stack<E> stack = new Stack<E>();

For example:

Stack<Integer> stack = new Stack<Integer>(); // It will store only Integer type objects.
```

### Stack Methods in Java

In addition to methods that inherit from vector class, stack class has five additional methods of its own. They are as follows:

**1. boolean empty():** This method is used to check the stack is empty or not. It returns true if and only if stack contains no elements (objects). Otherwise, it returns false.

**2. E peek():** This method is used to retrieve the top-most element from the stack without removing it.
**3. E pop():** The pop() method is used to pop (remove) the top-most element from the stack and returns it.
**4. E push(E obj):** This method pushes an element obj onto the top of the stack and returns that element (object).
**5. int search(Object obj):** This method returns the position of element obj from the top of stack. If the element is not present in the stack, it returns -1.

## Java Stack Example Programs

---

Let's take some example programs to perform the various operations based on the stack class methods in java.

**1. Adding Elements:** Using push() method, we can add elements to the stack. The push() method places the element at the top of stack.
**Program source code 1:**

```
import java.util.Stack;

public class StackEx {

public static void main(String[] args)

{

// Create an empty stack that contains String objects.

   Stack<String> st = new Stack<>();



// Checks that stack is empty or not.

   boolean empty = st.empty();
   System.out.println("Is stack empty: " +empty);



// Adds elements to the top of stack using push() method.

   st.push("Sunday");

   st.push("Monday");

   st.push("Tuesday");

   st.push("Wednesday");

   st.push("Thursday");

   st.push("Friday");

   st.push("Saturday");



// Displaying elements from the stack.

   System.out.println("Elements of Stack: " +st);

 }

}

Output:

      Is stack empty: true

      Elements of Stack: [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday]
```

**2. Accessing Element:** Using peek() method, we can retrieve or fetch an element at the top of stack without removing it. We will also serach the position of an element into the stack using search() method. Look at the source code below.

**Program source code 2:**

```
import java.util.Stack;

public class StackEx2 {

public static void main(String[] args)

{

// Create an empty stack that contains Integer objects.

  Stack<Integer> st = new Stack<>();



  st.push(25);

  st.push(30);

  st.push(35);

  st.push(40);

  st.push(45);

  st.push(50);



// Displaying elements from the stack.

  System.out.println("Elements of Stack: " +st);



// Retrieving element at the top of stack.
```

```
  Object peekElement = st.peek();

  System.out.println("Element at the top of stack: " +peekElement);



// Find the position of element into the stack.

  System.out.println("Position of element 50: " +st.search(50));

  System.out.println("Position of element 80: " +st.search(80));

 }

}

Output:

    Elements of Stack: [25, 30, 35, 40, 45, 50]

    Element at the top of stack: 50

    Position of element 50: 1

    Position of element 80: -1
```

**3. Removing Elements:** Using pop() method, we can pop an element from the stack in java.

**Program source code 3:**

```
import java.util.Stack;

public class StackEx3 {

public static void main(String[] args)

{

// Create an empty stack that contains Integer objects.
```

```
  Stack<Integer> st = new Stack<>();


  st.push(25);

  st.push(30);

  st.push(35);

  st.push(40);

  st.push(45);

  st.push(50);


// Displaying elements from the stack.

  System.out.println("Elements of Stack: " +st);


// Removing elements from the stack one by one.

  while(st.size() > 0){

          System.out.println("Removed element: " +st.pop());

  }

 System.out.println("Is stack empty: " +st.empty());

 }

}
```

Output:

```
Elements of Stack: [25, 30, 35, 40, 45, 50]

Removed element: 50

Removed element: 45

Removed element: 40

Removed element: 35

Removed element: 30

Removed element: 25

Is stack empty: true
```

**Program source code 4:**

```
import java.util.Stack;

public class StackEx4 {

public static void main(String[] args)

{

 Stack<Integer> st = new Stack<>();



   st.push(25);

   st.push(30);

   st.push(35);

   st.push(40);

   st.push(45);
System.out.println("Original elements of stack: " +st);

System.out.println("Pop element: " +st.pop());

System.out.println("Elements of stack after removing: " +st);



System.out.println("Push element: " +st.push(50));

System.out.println("Elements of stack after adding: " +st);

 }

}
```

Output:

```
    Original elements of stack: [25, 30, 35, 40, 45]

    Pop element: 45

    Elements of stack after removing: [25, 30, 35, 40]

    Push element: 50

    Elements of stack after adding: [25, 30, 35, 40, 50]
```

**Explanation:**

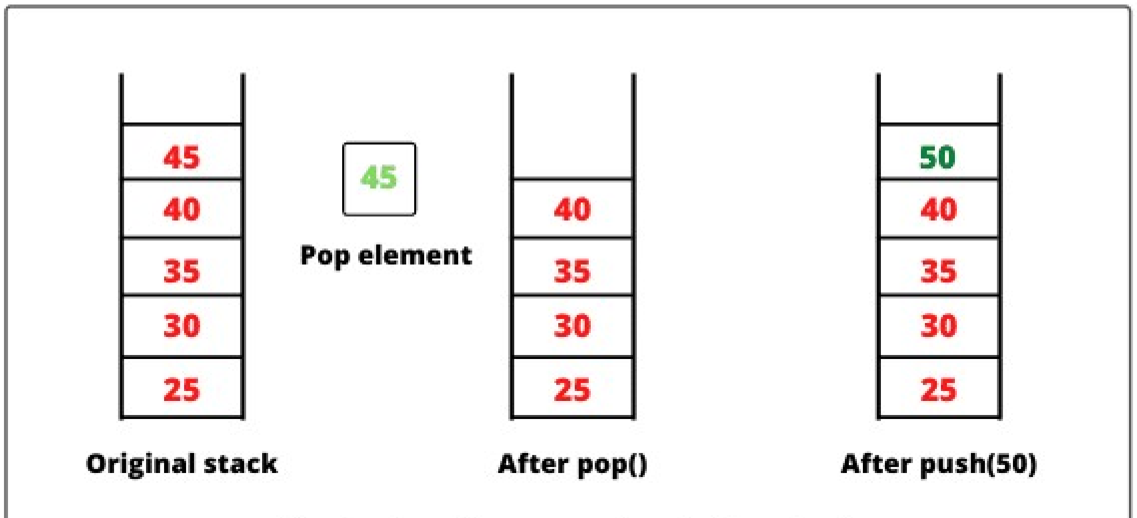


Fig: Push and Pop operations in Java Stack

Look at the above figure to understand push and pop operations in java stack more clearly.

## What is Hashtable in Java with Example

**Hashtable class in Java** is a concrete implementation of abstract Dictionary class.
It is a data structure similar to Java HashMap that can store a collection of elements (objects) in the form of key-value pairs (entries).
Key objects must implement hashCode() and equals() methods to store and retrieve values from the Hashtable.

In other words, Hashtable can only store key objects that override hashCode() and equals() methods defined by the Object class.

The main difference between Hashtable and HashMap is the way they work with thread access.

- Hashtable class is a synchronized class that means it is thread-safe. Multiple threads cannot access the same instance of the Hashtable class concurrently (at the same time).
- HashMap class is not synchronized which means it is not thread-safe. Multiple threads can access the same instance of HashMap

class simultaneously. Therefore, it is safe to use only when one thread uses an object.

Java Hashtable was added in JDK 1.0 version and present in java.util.Hashtable package. Prior to JDK 1.2 version, Hashtable was used for mapping keys with values.

Later on, Java 1.2 version modified Hashtable so that it implements Map interface. Thus, Hashtable has been integrated with Java collections framework.

But it is not quite consistent with the Java Collections Framework, it is now regarded as a "**legacy class**."

## Hierarchy of Java Hashtable

---

Hashtable class extends Dictionary class and implements Map, Cloneable, and Serializable interfaces. The hierarchy diagram of Hashtable in Java is shown in the below diagram.

# Java Map Interface Implementation Classes

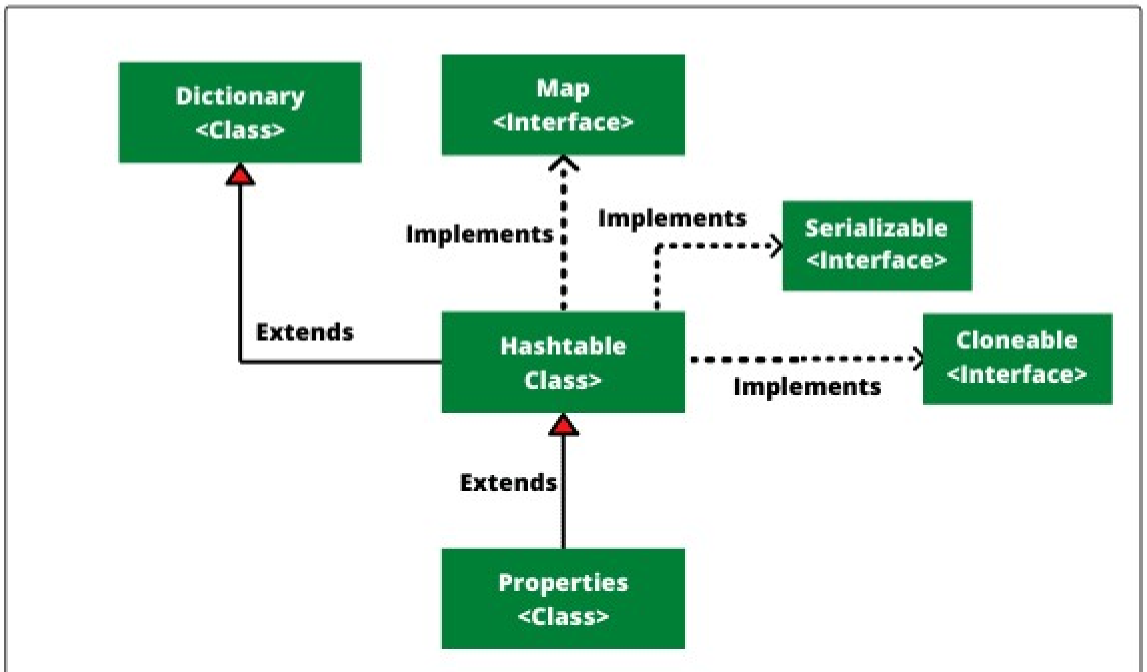


Fig: Hierarchy diagram of Hashtable in Java

## Hashtable class declaration

Hashtable in Java is a generic class made by JDK 1.5 that can be declared as follows:

```
public class Hashtable<K,V>

  extends Dictionary<K,V>

     implements Map<K,V>, Cloneable, Serializable
```

Here, Hashtable accepts two parameters K and V where K represents the type of keys and V defines the type of values.

## Features of Hashtable

# Java Map Interface Implementation Classes

There are several features of Hashtable in Java that must keep in mind to use it.

1. The underlying data structure for Java Hashtable is a hash table only.

2. Insertion order is not preserved. That means it does not maintain insertion order.

3. Duplicate keys are not allowed but values can be duplicated.

4. Heterogeneous objects are allowed for both keys and values.

5. Null is not allowed for both key and values. If we attempt to store null key or value, we will get a RuntimeException named NullPointerException.

6. Java Hashtable implements Serializable and Cloneable interfaces but not random access.

7. Every method present in Hashtable is synchronized. Hence, Hashtable object is thread-safe.

8. Hashtable is the best choice if our frequent operation is a retrieval (search) operation.

9. Since Hashtable is synchronized, its operations are slower as compared to HashMap in java.

## Constructors of Hashtable class in Java

Hashtable class defines the following constructors that are as follows:

**1. Hashtable():** This form of constructor constructs a new, empty hashtable object with a default initial capacity as 11 and load factor as 0.75.
To store a string as key and an integer object as its value, we can create a Hashtable object as:

```
Hashtable<String,Integer> ht = new Hashtable<>();
```

The default initial capacity for this hashtable object will be taken as 11 and load factor 0.75.

**2. Hashtable(int initialCapacity):** This constructor constructs a new, empty hashtable with the specified initial capacity and default load factor as 0.75. If the initial capacity is less than zero, it will throw an exception named IllegalArgumentException.
**3. Hashtable(int initialCapacity, float loadFactor):** This constructor constructs a new, empty hashtable with the specified initial capacity and the specified load factor.
**4. Hashtable(Map m):** This form of constructor constructs a new hashtable with the same mappings as the given Map. If the specified map is null, it will throw NullPointerException.

## Hashtable Methods in Java

---

Java Hashtable class defines the following legacy methods:

**1. void clear():** It is used to clear all key-value pairs from the Hashtable.
**2. Object clone():** This method is used to create a shallow copy of this hashtable.
**3. V put(K key, V value):** This method is used to map key-value pairs into the Hashtable.
**4. void putAll(Map t):** This method copies all of the key-value pairs (mappings) from the specified map into the Hashtable.
**5. boolean isEmpty():** It returns true if there are no entries in the Hashtable.

---

**6. int size():** This method is used to retrieve the number of key-value pairs in the Hashtable.

**7. V remove(Object key):** This method is used to remove the key (and its associated value) from the hashtable.
**8. boolean remove(Object key, Object value):** This method removes the entry for the specified key only if it is currently mapped to the specified value.
**9. V replace(K key, V value):** This method is used to replace the entry for the specified key only if it is currently mapped to some value.
**10. boolean replace(K key, V oldValue, V newValue):** This method replaces the key-value pair for the specified key only if currently mapped to the specified value.

---

**11. boolean contains(Object value):** This method checks Hashtable and returns true if hash table contains the specified value.
**12. boolean containsKey(Object key):** This method searches hash table and returns true if the specified key object in the hashtable.
**13. boolean containsValue(Object value):** This method searches hash table and returns true if this hashtable maps one or more keys to this value.
**14. Enumeration<V> elements():** This method returns an enumeration of the values in this hashtable.
**15. Enumeration<K> keys():** This method returns an enumeration of the keys in this hashtable.

---

**16. Set<Map.Entry<K,V>> entrySet():** This method returns a set or collection view of the mappings contained in this map.
**17. Set<K> keySet():** It returns a set view of the keys contained in this map.
**18. Collection<V> values():** It returns a collection view of the values contained in this map.

**19. V get(Object key):** It returns the value associated with its specified key. It returns null if the key does not have a value associated with it.
**20. boolean equals(Object o):** It is used to compare the specified Object for equality, as per the definition in the Map interface.

---

**21. int hashCode():** This method returns the value of hash code as per the definition in the Map interface.
**22. String toString():** This method is used to convert Hashtable objects into a string in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space) and returns it.

## Java Hashtable Example Programs

---

Let's take various example programs for performing operations based on methods of hash table in java.

1. Let's create a program where we will perform various operations such as adding, removing, checking hash table is empty or not before adding elements, and size.

In this example, we will create a constructor with default initial capacity of 11 and load factor 0.75. Look at the source code to understand better.

**Program source code 1:**

```
import java.util.Hashtable;

public class HashtableEx {

public static void main(String[] args)

{

// Create a Hashtable object.
```

```
  Hashtable<Integer, String> ht = new Hashtable<Integer, String>();


// Checking hashtable is empty or not.

   boolean isEmpty = ht.isEmpty();

   System.out.println("Is hash table empty: " +isEmpty);


// Adding entries using put() method in hash table.

  ht.put(1, "One"); // ht.size() is 1.

  ht.put(2, "Two"); // ht.size() is 2.

  ht.put(3, "Three"); // ht.size() is 3.

  ht.put(4, "Four"); // ht.size() is 4.

  ht.put(5, "Five"); // ht.size() is 5.

  ht.put(6, "Six"); // ht.size() is 6.


System.out.println("Displaying entries in hash table: " +ht);

int size = ht.size();

System.out.println("Size of hash table: " +size);


// Removing last entry.

  String removeE = ht.remove(6);
  System.out.println("Removed entry: " +removeE);

  System.out.println("Updated entries in hash table: " +ht);



// Getting the value of 4.

   String getValue = ht.get(4);



   System.out.println("Getting the value of 4: " +getValue);

   System.out.println("Getting the value of 2: " +ht.get(2));

  }

}
```

Output:

```
      Is hash table empty: true

      Displaying entries in hash table: {6=Six, 5=Five, 4=Four, 3=Three, 2=Two, 1=One}

      Size of hash table: 6

      Removed entry: Six

      Updated entries in hash table: {5=Five, 4=Four, 3=Three, 2=Two, 1=One}

      Getting the value of 4: Four

      Getting the value of 2: Two
```

As you can observe in the output of program, the insertion order is not retained in the hashtable. Internally, for every entry, a separate hash code of key is generated and the entries are indexed based on the hash code of keys to make it more efficient.

2. Let's create another program where we will perform various operations based on methods such as replace(), containsKey(), and containsValue().

**Program source code 2:**

```
import java.util.Hashtable;

public class HashtableEx2 {

public static void main(String[] args)

{

// Create a Hashtable object.

  Hashtable<String, Integer> ht = new Hashtable<>();



  ht.put("John", 20);

  ht.put("Shubh", 30);

  ht.put("Peter", 25);

  ht.put("Deep", 15);

  ht.put("Jonshan", 40);



System.out.println("Original entries of hash table: " +ht);



// Replacing an entry for specified key from hash table.
```

```
  Integer replace = ht.replace("Peter", 60);

  System.out.println("Replacing entry for specified key: " +replace);

  System.out.println("Updated entries in hash table: " +ht);



// Checking specified key present in hash table.

  boolean containsKey = ht.containsKey("Shubh");

  System.out.println("Is key Shubh in hash table: " +containsKey);



// Checking specified value present in hash table.

  boolean containsValue = ht.containsValue(40);

  System.out.println("Is value 40 in hash table: " +containsValue);

 }

}
```

Output:

```
    Original entries of hash table: {John=20, Jonshan=40, Shubh=30, Deep=15,
Peter=25}

    Replacing entry for specified key: 25

    Updated entries in hash table: {John=20, Jonshan=40, Shubh=30, Deep=15,
Peter=60}

    Is key Shubh in hash table: true

    Is value 40 in hash table: true
```

3. Let's take an example program where we will iterate keys, values, and entries of hash table using iterator(), keySet(), values(), and entrySet().

**Program source code 3:**

```
import java.util.Hashtable;

import java.util.Iterator;

import java.util.Map.Entry;



public class HashtableEx {

public static void main(String[] args)

{

Hashtable<String, Integer> ht = new Hashtable<>();



  ht.put("John", 20);

  ht.put("Shubh", 30);

  ht.put("Peter", 25);

  ht.put("Deep", 15);

  ht.put("Jonshan", 40);



System.out.println("Original entries of hash table: " +ht);



// Iterating elements of hash table using iterator() method.
```

```
   System.out.println("Iterating keys of hash table:");

   Iterator<String> itr = ht.keySet().iterator();

   while(itr.hasNext())

   {

    System.out.println(itr.next());

   }

 System.out.println("\n");



System.out.println("Iterating values of hash table:");

Iterator<Integer> itrValue = ht.values().iterator();

while(itrValue.hasNext())

{

    System.out.println(itrValue.next());

}

System.out.println("\n");

System.out.println("Iterating entries of hash table:");



Iterator<Entry<String, Integer>> itrEntry = ht.entrySet().iterator();

while(itrEntry.hasNext())

{
    System.out.println(itrEntry.next());

}

 }

}
```

Output:

```
     Original entries of hash table: {John=20, Jonshan=40, Shubh=30, Deep=15,
Peter=25}

     Iterating keys of hash table:

     John

     Jonshan

     Shubh

     Deep

     Peter



     Iterating values of hash table:

     20

     40

     30

     15

     25
```

```
Iterating entries of hash table

John=20

Jonshan=40

Shubh=30

Deep=15

Peter=25
```

## Properties in Java | Example Program

**Properties in Java** is a class that is a child class (subclass) of Hashtable.
It is mainly used to maintain the list of values in which key and value pairs are represented in the form of strings.
In other words, keys and values in Properties objects should be of type String.

Properties class was added in JDK 1.0 version. It is present in java.util.Properties package. It provides additional methods to retrieve and store data from the properties file.

### Hierarchy of Properties class in Java

Properties class extends Hashtable class that extends Dictionary class. It implements Map, Serializable, and Cloneable interfaces.

The hierarchy diagram of Properties class in Java is shown in the below figure.

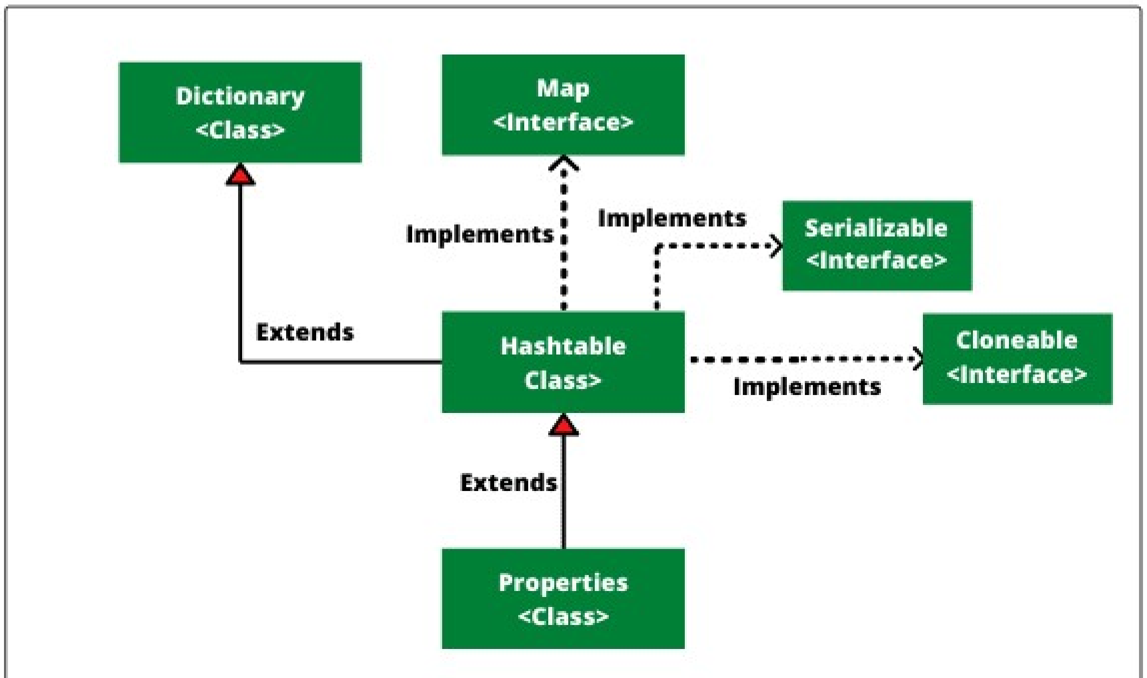


Fig: Hierarchy diagram of Hashtable in Java

## Properties class declaration

The general declaration of Properties class is declared as given below:

```
public class Properties

   extends Hashtable<Object,Object>
```

## Variable defined by Properties class

Java Properties class defines the following instance variable. The general syntax is as follows:

```
Protected Properties defaults;
```

This instance variable stores a default properties list associated with a Properties object.

## Constructors of Properties class

---

Java Properties class defines the following constructors that are as follows:

**1. Properties():** This form of a constructor is used to create an empty property list (map) with no default values. The general syntax to create Properties object is as follows:

```
Properties p = new Properties();
```

**2. Properties(Properties defaults):** This form of constructor creates an empty property list with the specified defaults. The general syntax to create Properties object with the specified defaults is as follow:

```
Properties p = new Properties(Properties defaults);
```

## Methods of Properties class in Java

---

In addition to methods that Properties class inherit from Hashtable, Properties class defines some more additional methods that are as:

**1. String getProperties(String key):** This method returns the value associated with key. If the key is neither in the list nor in the default property list, it will return null object.
**2. String getProperty(String key, String defaultValue):** This method returns the value associated with key. If the key is neither in the list nor in the default property list, it will return defaultValue.

**3. void list(PrintStream streamOut):** This method prints the property list streamOut to the specified output stream.
**4. void list(PrintWriter streamOut):** This method prints this property list streamOut to the specified output stream.
**5. void load(InputStream inStream):** This method loads (reads) a property list (key and element pairs) from the InputStream.

---

**6. void load(Reader reader):** This method loads a property list (key and element pairs) from the input character stream in a simple line-oriented format.
**7. void loadFromXML(InputStream in):** This method loads all of the properties represented by the XML document on the specified InputStream into the properties table.
**8. Enumeration propertyNames():** It returns an enumeration of all the keys in this property list, including distinct keys in the default property list.
**9. Object setProperties(String key, String value):** It returns value associated with key. It returns null if value associated with key does not exist.

**10. void store(OutputStream streamOut, String description):** It is used to store a property list to an OutputStream.

## How to get Data from Properties file using Properties class?

---

1. First, create a properties file and store data into it. Save it as db.properties.

```
username = John
```

```
password = 12345
```

2. Now, creates java class to read the data from the properties file.

**Program source code 1:**

```
import java.util.*;

import java.io.*;

public class Test {

public static void main(String[] args)throws Exception {

  FileReader reader=new FileReader("db.properties");



   Properties p=new Properties();

   p.load(reader);



 System.out.println(p.getProperty("username"));

 System.out.println(p.getProperty("password"));

   }

}

Output:

       John

       12345
```

## How to get all System properties using Properties class in Java?

# Java Map Interface Implementation Classes

By using System.getProperties() method of Properties class, we can get all the properties of the system. Let’s create a program where we will get data from the system properties. Look at the source code to understand better.

**Program source code 2:**

```
import java.io.IOException;

import java.util.Iterator;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Properties;

import java.util.Set;



public class Test {

public static void main(String[] args) throws IOException

{

 Properties p = System.getProperties();

  Set<Entry<Object, Object>> set = p.entrySet();



 Iterator<Entry<Object, Object>> itr = set.iterator();

 while(itr.hasNext())
 {
 Map.Entry entry = (Map.Entry)itr.next();
 System.out.println(entry.getKey()+" = "+entry.getValue());
 }
 }
}
Output:
     java.runtime.name = Java(TM) SE Runtime Environment
     sun.boot.library.path = C:\Program Files (x86)\Java\jre1.8.0_181\bin
     java.vm.version = 25.181-b13
     java.vm.vendor = Oracle Corporation
     java.vendor.url = http://java.oracle.com/
     path.separator = ;
     java.vm.name = Java HotSpot(TM) Client VM
     file.encoding.pkg = sun.io
     user.country = IN
     user.script =
     sun.java.launcher = SUN_STANDARD
     sun.os.patch.level =
     java.vm.specification.name = Java Virtual Machine Specification
 . . . . . . . . . . . . . . . . . .

.. . . . . . . . . . . . . . . . . . .
```

## How to create Properties file using Properties class?

---

Let’s take an example program where we will create properties file using java properties class. Look at the source code.

**Program source code 3:**

```
import java.io.FileWriter;

import java.io.IOException;

import java.util.Properties;



public class Test {

public static void main(String[] args) throws IOException

{

 Properties p = new Properties();

   p.setProperty("Name","John");

   p.setProperty("Email","john@scientecheasy.com");



FileWriter fw = new FileWriter("info.properties");

 p.store(fw,"Scientech Easy Properties File Example");
 }

}
```

Now refresh your project folder and see the properties file named "info.properties". Open info.properties file to see the result.

```
Output:

     #Scientech Easy Properties File Example

     #Thu Dec 17 09:24:58 IST 2020

     Name=John

     Email=john@scientecheasy.com
```

### Advantage of  Properties file

The main advantage of the properties file is that recompilation is not needed if any data is changed from a properties file.

If any data is changed from the properties file, we don't require to recompile the java class. It is used to store data that is to be changed frequently.

## Vector in Java | Methods, Example

**Vector class in Java** was introduced in JDK 1.0 version. It is present in Java.util package. It is a dynamically resizable array (growable array) which means it can grow or shrink as required.
Java Vector class is similar to ArrayList class with two main differences.

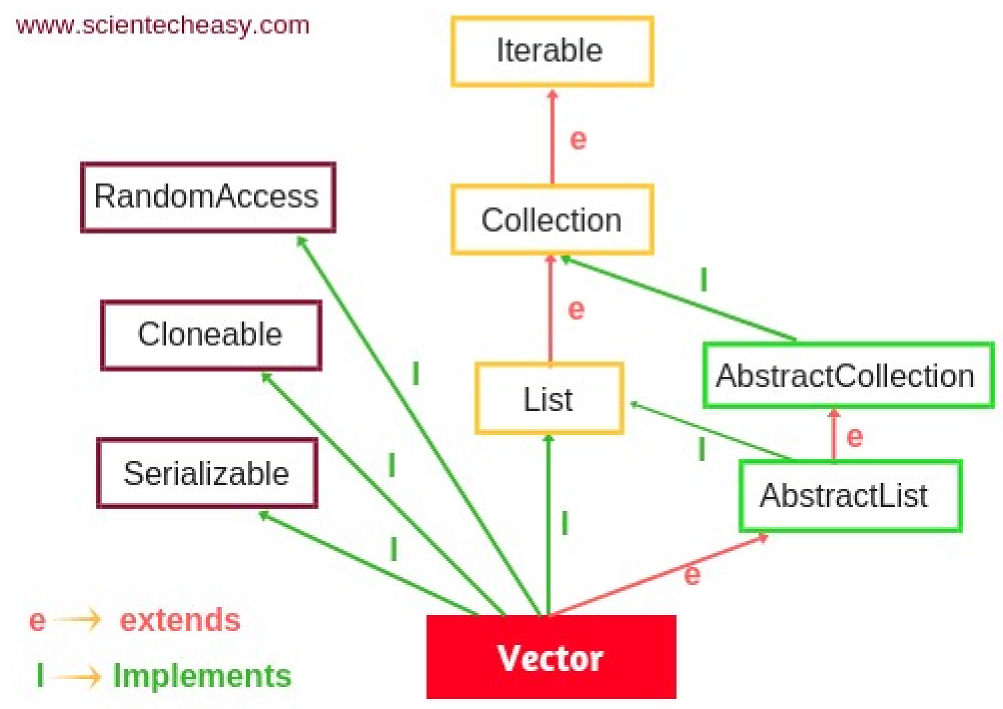
- Vector is synchronized. It is used for thread safety.
- It contains many legacy methods that are not now a part of the collections framework.

# Java Map Interface Implementation Classes

## Hierarchy Diagram of Vector class in Java

---

Vector class implements List interface and extends AbstractList. It also implements three marker interface such as serializable, cloneable, and random access interface.
The current hierarchy diagram of a Vector class is shown in the below figure.



Hierarchy diagram of Vector

## Features of Vector class

---

1. The underlying Data structure for vector class is the resizable array or growable array.

2. Duplicate elements are allowed in the vector class.

3. It preserves the insertion order in Java.

4. Null elements are allowed in the Java vector class.

5. Heterogeneous elements are allowed in the vector class. Therefore, it can hold elements of any type and any number.

6. Most of the methods present in the vector class are synchronized. That means it is a thread-safe. Two threads cannot access the same vector object at the same time. Only one thread can access can enter to access vector object at a time.

7. Vector class is preferred where we are developing a multi-threaded application but it gives poor performance because it is thread-safety.

8. Vector is rarely used in a non-multithreaded environment due to synchronized which gives you poor performance in searching, adding, delete, and update of its element.

9. It can be iterated by a simple for loop, Iterator, ListIterator, and Enumeration.

10. Vector is the best choice if the frequent operation is retrieval (getting).

## Java Vector Class Constructors

---

Vector class provides four types of constructors to create, access, and modify the data structure. They are listed below in table form.

| Constructor | Description |
|---|---|
| vector() | It creates an empty vector with default initial capacity of 10. |

| | |
|---|---|
| vector(int initialCapacity) | It creates an empty vector with specified initial capacity. |
| vector(int initialCapacity, int capacityIncrement) | It creates an empty vector with specified initial capacity and capacity increment. |
| vector(Collection c) | It creates a vector that contains the element of collection c. |

## Ways to create Vector class object in Java

There are four ways to create an object of vector class in Java. They are as follows:

```
1. Vector vec = new Vector();
```

It creates a vector list with a default initial capacity of 10. Inside the vector list, data is stored as an array of objects according to the initial size that you specify in the constructor. Since default initial capacity is 10. So, we can store only 10 elements.

When the 11th element is inserted into vector, the capacity of vector will be exceeded. In this case. a new internal array will be automatically created that is equal to the size of old array plus capacity increment specified in the constructor.

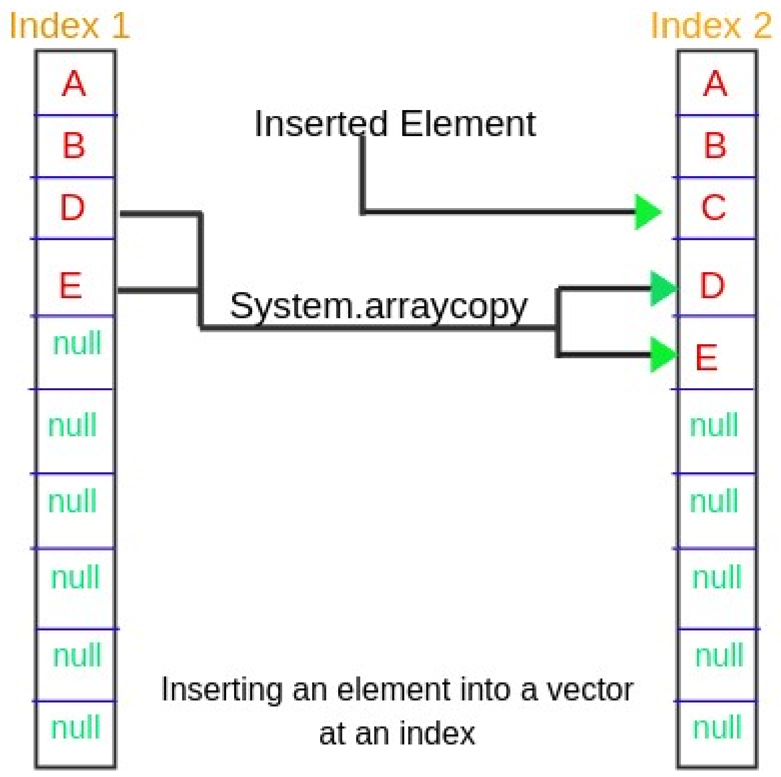
If you do not specify an initial capacity and capacity increment, each time vector will be resized due to exceeding its capacity. A new internal array will create which copies all the original elements of old array into the new array by using System.arraycopy() method.

Creating a new array and copying elements will take more time. Therefore, it is important to assign the value of initial capacity and capacity increment for minimizing the number of resizing operations without wasting too much memory and time.

In this object creation, we did not specify any capacity increment in the constructor. So, its size will be double by default that will be 20.

After resizing, it copies all original elements of the old array into the new array using System.arraycopy() method, and then it will insert 11th element.

When we add an element at a specific position, the vector uses a System.arraycopy() method to move the element from that specific position to the one position down into the array. Look at the below figure to understand better.



Similarly, if we remove an element at a specific position, the same process is performed by vector and elements will be shifted one position up.

But vector is not a good choice in inserting and removal of elements because it will take more time to copy elements into new array and will get slower as the size of the vector grows.

```
2. Vector vec = new Vector(int initialCapacity);
```

```
For example:

    Vector vec = new Vector(3); // It will create an empty vector with initial capacity of 3.



3. Vector v = new Vector(int initialCapacity, capacityIncrement);



For example:

    Vector v = new Vector(3, 5);



4. Vector v = new Vector(Colletion c);



For example:

      Vector v = new Vector(list1); // list1 is elements of the collection.
```

## Code to create Generic Vector class object

We can also create a vector object in generic form like this:

```
Vector<T> vec = new Vector<T>(); // T is the type of generic.



For example:

    Vector<String> vec = new Vector<String>(); // It will store only string type element.
```

```
Vector<Integer> v = new Vector<Integer>(3,6);
```

## Vector Methods in Java

Vector class provides many different methods to create, access, and modify the data structure. They are as follows:

**1. boolean add(Object o):** It is used to add the specified element to the end of given vector list. It will return true if the specified element is added successfully into vector list otherwise, it will return false.
For example: v.add("John");

**2. void add(int index, Object o):** This method is used to insert a specified element at a specific position into the vector list. It returns nothing.
For example: v.add(4, "John"); // It adds the element 'John' at position 4 in the vector.

**3. boolean addAll(Collection c):** It adds a collection of elements to a vector. It returns true if the collection of elements is successfully added in the vector otherwise, false. If the collection is null, it will throw NullPointerException.
For example: v.addAll(list2);

**4. boolean addElement(Object o):** It adds an element to the end of a vector. It increases the vector size by one.
For example: v.addElement("Cat");

**5. int capacity():** It is used to get the current capacity of a vector or length of array present in the vector. Its return type is Integer value. This method does not take any parameter.
For example: v.capacity();

**6. int size():** It returns the number of elements in a vector.

**Note:** The difference between size() and capacity() is that size() is the number of elements that are currently held whereas capacity() is the number of elements that can maximum hold.

## Java Vector Class Example Programs

---

Let’s take an example program based on the above vector methods.

**Program source code 1:**

```
package vectorTest;

import java.util.ArrayList;

import java.util.Vector;

public class VectorTest1

{

public static void main(String[] args)

{

// Create an empty vector object with an initial capacity of 5.

   Vector v = new Vector();



// Check the size of vector before adding elements.

   int size = v.size();

   System.out.println("Size of vector before adding elements: " +size);
```

```
// Adding elements to a vector using reference variable v.

    v.add("Red");

    v.add("Green");

    v.add("Yellow");

    v.add("Pink");

    v.add("White");

  System.out.println("Vector elements: " +v);



// Check size and capacity.

   int vectorsize = v.size();

   System.out.println("Size: " +vectorsize);



   int capacity = v.capacity();

   System.out.println("Default capacity: " +capacity);



// Adding elements using addElement() method.

    v.addElement(null);

    v.addElement(20);

    v.add(5, 15);

  int newsize = v.size();
  System.out.println("New size after adding elements: " +newsize);


int newcapacity = v.capacity();

System.out.println("New capacity after adding elements: " +newcapacity);

System.out.println("Elements order after addition: " +v);


// Create an ArrayList with an initial capacity of 10.

   ArrayList al = new ArrayList();


// Adding elements using reference variable al.

    al.add("Brown");

    al.add("Black");


// Call addAll() method to add collection of elements into vector.

    v.addAll(al);


// Now check size and capacity of the vector.

   int vsize = v.size();

   System.out.println("Size: " +vsize);

   int vcapacity = v.capacity();
   System.out.println("Vcapacity: " +vcapacity);


// Adding 11th element to check size and capacity.

   v.add(10);

   System.out.println("Elements: " +v);


   int vecsize = v.size();

   System.out.println("Size after adding 11th element: " +vecsize);


    int cap = v.capacity();

    System.out.println("Capacity after adding 11th element: " +cap);

  }

}
```

Output:

```
      Size of vector before adding elements: 0

      Vector elements: [Red, Green, Yellow, Pink, White]

      Size: 5

      Default capacity: 10

      New size after adding elements: 8

      New capacity after adding elements: 10
```

```
Elements order after addition: [Red, Green, Yellow, Pink, White, 15, null, 20]

Size: 10

Vcapacity: 10

Elements: [Red, Green, Yellow, Pink, White, 15, null, 20, Brown, Black, 10]

Size after adding 11th element: 11

Capacity after adding 11th element: 20
```

In this example program, you can see that when we added the 11th element into vector, its capacity becomes double.

**7. void clear():** It clears or removes all elements from a vector.
For example: v.clear();

**8. Object clone():** It creates a clone of a vector.
For example: v.clone();

**9. boolean contains(Object o):** It will return true if the vector contains the specified element.
For example: v.contains("A");

**10. void copyInto(Object[ ] anArray):** It copies the elements of a vector into a specified array.
For example: v.copyInto(arr); // arr is the reference variable of specified array object.

**11. Object elementAt(int index):** It returns an element at a specific position.
For example: v.elementAt(2);

**12. Object firstElement():** It returns the element stored at index position 0.
For example: v.firstElement();

Let's take an example program based on the above vector methods.

**Program source code 2:**

```
package vectorTest;

import java.util.Vector;

public class VectorTest2

{

public static void main(String[] args)

{

// Create an empty generic vector with an initial capacity of 10.

   Vector<String> v = new Vector<String>();



// Adding elements to vector.

    v.add("A");

    v.add("B");

    v.add("C");

    v.add("D");

    v.add("E");

  System.out.println("Elements: " +v);



// Call firstElement() method to get the first element using reference variable v.

// Since the return type of firstElement method is String. Therefore, we will store it using
variable firstElement of data type String.
```

```
   String firstElement = v.firstElement();

   System.out.println("First Element: " +firstElement);



String lastElement = v.lastElement();

System.out.println("Last Element: " +lastElement);



String element = v.elementAt(3);

System.out.println("Element at position 3: " +element);



boolean checkElement = v.contains("F"); // Return type of contains method is boolean.

System.out.println("Element F: " +checkElement);



// Create an array object with an initial capacity of 5.

   String[] arr = new String[5];



// Copy collection of elements into an array str.

    v.copyInto(arr);

    System.out.println("Elements in an array arr");

    for(String str:arr)

    {
     System.out.println(str);

    }



// Call clone() method to create a clone of a vector.

    Object obj = v.clone(); // Return type of clone method is an Object.

    System.out.println("Clone of v: " +obj);

  }

}

Output:

     Elements: [A, B, C, D, E]

     First Element: A

     Last Element: E

     Element at position 3: D

     Element F: false

     Elements in an array arr A B C D E

     Clone of v: [A, B, C, D, E]
```

**13. Object get(int index):** This method is used to get an element at a specific position in the vector.
For example: v.get(2); // It will return an element from position 2.

**14. int hashCode():** This method returns hash code value for a vector.

**15. int indexOf(Object o):** It returns the index of the first occurrence of a specified element in the vector. It returns -1 if the vector does not contain the element.
For example: v.indexOf("C");

**16. int lastIndexOf(Object o):** It returns the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
**17. void insertElementAt(Object o, int index):** It inserts an element into the vector.
For example: v.insertElementAt("C", 4); // Vector class will insert element C at index 4.

**18. boolean remove(Object o):** It removes or clears the first occurrence of a specific element in the vector.
**19. boolean removeAll(Collection c):** It removes a collection of elements from the vector.
**20. void removeAllElements():** It deletes all elements from the vector and sets the size of the vector to zero.
**21. boolean removeElement(Object o):** It clears the first occurrence of a specific element from a vector.
**22. void removeElementAt(int index):** It removes an element at a specific position from the vector.
Let's take an example program where we will implement these vector methods.

**Program source code 3:**

```
package vectorTest;

import java.util.Vector;

public class VectorTest3

{

public static void main(String[] args)
```

```
{

 Vector v = new Vector();

  v.add("Bat");

  v.add("Ball");

  v.add("Wicket");

  v.add("Stump");

  v.add("Pitch");

  v.add(25);

  v.add(null);



System.out.println("Elements: " +v);

Object getElement = v.get(5); // Return type of get method is an Object.

System.out.println("Element at position 5: " +getElement);



Integer hashcode = v.hashCode(); // Return type is an Integer.

System.out.println("HashCode value: " +hashcode);



Integer indexOfElement = v.indexOf(null);

System.out.println("Index of element null: " +indexOfElement);
```

```
  v.insertElementAt("Gloves", 6);

  v.removeElement(25);

  v.remove(6);

System.out.println("Elements after removing: " +v);

 }

}

Output:

      Elements: [Bat, Ball, Wicket, Stump, Pitch, 25, null]

      Element at position 5: 25

      HashCode value: 461290222

      Index of element null: 6

      Elements after removing: [Bat, Ball, Wicket, Stump, Pitch, Gloves]
```

**23. boolean isEmpty():** It is used to check if the vector is empty. If the vector is empty, it will return true otherwise, false.
**24. void removeRange(int fromIndex, toIndex):** It is used to remove a range of elements from a vector. This method does not return any value. It accepts two parameters.
**fromIndex:** Starting index from where you have to remove.
**toIndex:** the last index as far as you have to remove all elements.
For example: v.removeRange(3, 6);

**25. void set():** It is used to change the element at a specified position with the specified element in the vector.
**26. void setElementAt(Object o, int index):** It is used to change an element at a specific position with a specified element within the vector.

For example: v.setElement(“B”, 4);

**27. void setSize(int newSize):** It is used to change the size of an internal vector buffer.

**28. boolean equals(Object o):** It is used to check the equality with another object or element. It returns true if the element is matched.

For example: v.equals(v1);

**Program source code 4:**

```
package vectorTest;

import java.util.Vector;

public class VectorTest4

{

public static void main(String[] args)

{

 Vector<String> vec = new Vector<String>();

// Check vector is empty or not.

  boolean check = vec.isEmpty();

  System.out.println("Vector is empty: " +check);



  vec.add("Hydrogen");

  vec.add("Oxygen");

  vec.add("Boron");

  vec.add("Beryllium");
  vec.add("Boron");

 System.out.println("Elements: " +vec);



 boolean check1 = vec.isEmpty();

 System.out.println("Vector is empty: " +check1);



// Replace element oxygen with helium.

   vec.setElementAt("Helium", 1);

   vec.set(2, "Lithium");

  System.out.println("Elements after replacing: " +vec);



// Check size of the vector.

   int size = vec.size();

   System.out.println("Size of the vector: " +size);



// Setting new size of the vector.

   vec.setSize(8);

   System.out.println("Size of the vector after setting: " +vec.size());



// Check capacity of the vector.
```

```
  int capacity = vec.capacity();

  System.out.println("Capacity of the vector: " +capacity);



// Check the equality of element carbon.

  boolean equality = vec.equals("Carbon");

  System.out.println("Is Carbon present: " +equality);

 }

}
```

Output:

```
    Vector is empty: true

    Elements: [Hydrogen, Oxygen, Boron, Beryllium, Boron]

    Vector is empty: false

    Elements after replacing: [Hydrogen, Helium, Lithium, Beryllium, Boron]

    Size of the vector: 5

    Size of the vector after setting: 8

    Capacity of the vector: 10

    Is Carbon present: false
```

**29. void ensureCapacity(int minCapacity):** This method is used to increase the capacity of the vector at a certain size.
For example: v.ensureCapacity(15);

**30. void trimToSize():** It is used to trim the capacity of the vector to the vector's actual size.
**31. String toString():** It converts vector contents into a string. It returns a string representation of the vector.
**32. object[ ] toArray():** It returns the elements of a vector as an array.
**Program source code 5:**

```
package vectorTest;

import java.util.Vector;

public class VectorTest5

{

public static void main(String[] args)

{

 Vector<Integer> v = new Vector<Integer>();

 for(int i = 0; i < = 10; i++)

 {

  if(i % 2 == 0)

  {

    v.add(i);

    System.out.println(i);

   }

 }

 int size = v.size();
 System.out.println("Size of the vector: " +size);

 int capacity = v.capacity();

 System.out.println("Capacity of the vector: " +capacity);



// Ensuring capacity.

   v.ensureCapacity(25);

// Checking capacity.

   System.out.println("Minimum capacity: " +v.capacity());



// Trim the capacity of the vector to the actual size.

   v.trimToSize();

   System.out.println("Minimum capacity after trimming: " +v.capacity());



// String representation of the vector.

   String str = v.toString();

   System.out.println("String equivalent of the vector: " +str);



// Get elements of vector as an array form.

    v.toArray();

    System.out.println(v);
```

```
  }

 }

Output:

      0 2 4 6 8 10

      Size of the vector: 6

      Capacity of the vector: 10

      Minimum capacity: 25

      Minimum capacity after trimming: 6

      String equivalent of the vector: [0, 2, 4, 6, 8, 10]

      [0, 2, 4, 6, 8, 10]
```

## When to Use Vector?

The vector can be used when
➤ We want to store duplicate and null elements.

➤ We are developing a multi-threaded application but it can reduce the performance of the application because it is “thread-safety”.

➤ Vector is more preferred when the retrieval of elements is more as compared to insertion and removals of elements.

➤ Vector is a good choice if you want to access the data in the list rapidly and a poor choice if the data in the list is modified frequently.

## Difference between ArrayList and Vector in Java

# Java Map Interface Implementation Classes

| ArrayList | Vector |
|---|---|
| ArrayList is not synchronized. | Vector is synchronized. |
| Since ArrayList is not synchronized. Hence, its operation is faster as compared to vector. | Vector is slower than ArrayList. |
| ArrayList was introduced in JDK 2.0. | Vector was introduced in JDK 1.0. |
| ArrayList is created with an initial capacity of 10. Its size is increased by 50%. | Vector is created with an initial capacity of 10 but its size is increased by 100%. |
| In the ArrayList, Enumeration is fail-fast. Any modification in ArrayList during the iteration using Enumeration will throw ConcurrentModificatioException. | Enumeration is fail-safe in the vector. Any modification during the iteration using Enumeration will not throw any exception. |

## Vector Programs in Java for Best Practice

In this tutorial, we have listed various types of Vector programs in Java for practice based on Vector methods, Enumeration, Iterator, and ListIterator.

Before going to practice, I will recommend going to the previous vector tutorial for clear concepts.

### Java Vector Program based on Enumeration

Let's create a program where we will iterate elements of vector list using Enumeration. We will also add or remove an element during the iteration. It will not throw ConcurrentModificatioException. Follow all the steps in the given source code.

**Program source code 1:**

```
package vectorTest;
```

## Java Map Interface Implementation Classes

```
import java.util.Enumeration;

import java.util.Vector;

public class VectorEnumerationEx

{

public static void main(String[] args)

{

// Create vector object of type String.

   Vector<String> v = new Vector<String>();



// Adding elements to vector using add() method of vector class.

    v.add("A"); // Adding element at index 0.

    v.add("B"); // Adding element at index 1.

    v.add("C"); // Adding element at index 2.

    v.add("D"); // Adding element at index 3.

    v.add("E"); // Adding element at index 4.



// Call elements() method to iterate vector.

   Enumeration<String> en = v.elements(); // Return type is Enumeration.

   System.out.println("Vector elements are: ");
// Checking the next element availability using reference variable en in the while loop.

   while(en.hasMoreElements())

   {

// Moving cursor to the next element.

    Object obj = en.nextElement(); // Return type is Object.

    System.out.println(obj);



// Adding and removing an element during iteration using Enumeration.

// Enumeration is fail-safe in a vector. So, it will not throw any exception.

    v.removeElementAt(4);

    v.add(4, "G");

  }

  System.out.println("Vector list after adding elements during Iteration");

  System.out.println(v);

 }

}

Output:

       Vector elements are: A B C D G

       Vector list after adding elements during Iteration

       [A, B, C, D, G]
```

**Vector Program based on Iterator**

Let's create a program where we will iterate elements of vector in forwarding direction using Iterator. When we add or remove an element during the iteration, it will throw **ConcurrentModificationException**. Look at the source code.

**Program source code 2:**

```
package vectorTest;

import java.util.Iterator;

import java.util.Vector;

public class VectorIteratorExample

{

public static void main(String[] args)

{

  Vector<Integer> v = new Vector<Integer>();

   v.add(20);

   v.add(30);

   v.add(40);

   v.add(50);

   v.add(60);



// Call iterator() method to iterate elements of vector.
```

```
  Iterator<Integer> itr = v.iterator(); // Return type is Iterator.

  System.out.println("Vector elements are: ");



// Checking the next element availability in the list.

  while(itr.hasNext())

  {

// Moving cursor to the next element.

   Object obj = itr.next(); // Return type is Object.

   System.out.println(obj);



// Adding and removing an element during iteration using iteration. It will throw
ConcurrentModificationException. Since, Iterator is fail-fast in vector.

   v.add(4, 70);

 }

 System.out.println("Vector list after adding elements during Iteration");

 System.out.println(v);

 }

}

Output:

     Vector elements are: 20
     Exception in thread "main" java.util.ConcurrentModificationException at
java.util.Vector$Itr.checkForComodification(Vector.java:1210) at

     java.util.Vector$Itr.next(Vector.java:1163) at
vectorTest.VectorIteratorExample.main(VectorIteratorExample.java:29)
```

**Vector ListIterator Program in Java**

Let’s make a program where we will iterate elements of vector list in both forward and backward direction using ListIterator.

Using ListIterator, we can iterate a vector in both forward as well as backward direction. Look at the below program code.

**Program source code 3:**

```
package vectorTest;

import java.util.ListIterator;

import java.util.Vector;

public class VectorListIteratorEx

{

public static void main(String[] args)

{

 Vector<String> v = new Vector<String>();

  v.add("a");

  v.add("ab");

  v.add("abc");

  v.add("abcd");
 v.add("abcde");



// Call listIterator() method to iterate in the forward direction.

  ListIterator<String> litr = v.listIterator();

  System.out.println("Traversing in Forward Direction ");

  while(litr.hasNext())

  {

   Object obj = litr.next(); // Return type is Object.

   System.out.println(obj);

  }

 ListIterator<String> litr1 = v.listIterator(3); // It will iterate from index position 3 in the
backward direction.

 System.out.println("Traversing in Backward Direction");

 while(litr1.hasPrevious())

 {

  System.out.println(litr1.previous());

  }

 }

}

Output:
    Traversing in Forward Direction

    a ab   abc abcd abcde

    Traversing in Backward Direction

    abcd abc ab a
```

**How to Sort Vector Elements in Java?**

We know that vector maintains the insertion order that means it displays the same order in which they got added elements to the vector but we can also sort vector elements in the ascending order using Collections.sort() method. Let’s see an example program to sort vector elements of the collection object.

**Program source code 4:**

```
package vectorTest;

import java.util.Collections;

import java.util.Vector;

public class SortingVectorEx

{

public static void main(String[] args)

{

  Vector<String> v = new Vector<String>();

   v.add("Cricket");

   v.add("Football");

   v.add("Hockey");
   v.add("Volleyball");

   v.add("Basketball");



// By default vector maintains the insertion order.

   System.out.println("Vector elements before sorting:");

   for(int i = 0; i < v.size(); i++)

   {

// Call get() method to fetch elements from index and pass the parameter i.

   Object obj = v.get(i); // Return type of get() method is an Object.

   System.out.println(obj);

   }

// Now call Collections.sort() method to sort elements of vector in the ascending order.

   Collections.sort(v);



// Display vector elements after sorting.

   System.out.println("Vector elements after sorting:");

   for(int i = 0; i < v.size(); i++)

   {

   Object elements = v.get(i);

   System.out.println(elements);
   }

  }

}

Output:

       Vector elements before sorting:

       Cricket Football Hockey Volleyball Basketball

       Vector elements after sorting:

       Basketball Cricket Football Hockey Volleyball
```

**How to get SubList of Vector Elements in Java?**

We can get a sublist of vector elements using subList() method of vector class. This method will return a range of elements from the list between fromIndex, inclusive, and toIndex, exclusive.

The general syntax of subList() method is as follows:

```
public List subList(int fromIndex, int toIndex)
```

Let's create a program where we will get the sublist of elements of vector list. Look at the program source code to understand better.

**Program source code 5:**

```
package vectorTest;

import java.util.List;

import java.util.Vector;

public class SublistExample

{

public static void main(String[] args)
```

```
{

 Vector<String> v = new Vector<String>();

  v.add("Element1");

  v.add("Element2");

  v.add("Element3");

  v.add("Element4");

  v.add("Element5");

  v.add("Element6");



// Call subList() method to get a range of elements from the list.

   List<String> subList = v.subList(1, 5);

   System.out.println("Sub list elements:");

  for(int i = 0 ; i < subList.size() ; i++)

  {

// Call get() method to fetch elements from index and pass the parameter i.

   Object obj = subList.get(i); // Return type of get() method is an Object.

   System.out.println(obj);

  }

// We can also remove a range of elements using subList() method like this:

   System.out.println("List of elements after removing:");
```

```
    v.subList(3, 5).clear();

   System.out.println(v);

  }

}

Output:

       Sub list elements:

       Element2 Element3 Element4 Element5

       List of elements after removing:

       [Element1, Element2, Element3, Element6]
```

**Convert Vector to List and ArrayList**

We can convert a Vector into List using Collections.list(vector.elements()) method. This method will return a list of elements. Let's see a simple example to understand concepts.

**Program source code 6:**

```
package vectorTest;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.Vector;



public class ConversionExample

{
```

```
public static void main(String[] args)

{

  Vector<String> v = new Vector<String>();

   v.add("Element1");

   v.add("Element2");

   v.add("Element3");

   v.add("Element4");

  System.out.println("Vector Elements:");

  for (String str : v)

  {

    System.out.println(str);

  }

// Converting Vector to List.

   List<String> list = Collections.list(v.elements());

   System.out.println("List Elements:");

 for (String str2 : list)

  {

   System.out.println(str2);

  }

// Converting Vector to ArrayList.
```

```
  ArrayList<String> al = new ArrayList<String>(v);

  System.out.println("ArrayList Elements :");

  for (String s : al)

  {

   System.out.println(s);

  }

 }

}

Output:

    Vector Elements:

    Element1 Element2 Element3 Element4

    List Elements: Element1 Element2 Element3 Element4

    ArrayList Elements : Element1 Element2 Element3 Element4
```

## How to convert Vector to String array in Java?

Using toString() method of a Vector class, we can convert a Vector of Strings to an array. The syntax for toString() method is given below:

```
public String toString(); // It returns a string representation of each element of vector.
```

Let’s see an example program based on it.

**Program source code 7:**

```
package vectorTest;

import java.util.Vector;
```

```
public class VectorToArrayExample

{

public static void main(String[] args)

{

 Vector<String> v = new Vector<String>();

  v.add("Element1");

  v.add("Element2");

  v.add("Element3");

  v.add("Element4");

  v.add("Element5");


// Call size() method to get the size of vector.

   int size = v.size();

   System.out.println("Size of Vector: " +size);


// Converting vector to Array.

// Create an object of String array and pass the parameter size.

   String[] str = new String[size];


// Call toArray() method to get an array representation of the elements of this vector.
```

```
   String[] arrayOfElements = v.toArray(str); // It returns an array containing all the
elements of this vector.



// Displaying Array Elements.

   System.out.println("String Array Elements:");

  for(int i = 0; i < arrayOfElements.length; i++)

  {

    System.out.println(arrayOfElements[i]);

   }

  }

}

Output:

       Size of Vector: 5

       String Array Elements:

       Element1 Element2 Element3 Element4 Element5
```

In the above program source code, line number 20 and 22 can be also written in one step like this:

```
String[ ] arrayOfElements = v.toArray(new String[v.size()]);
```

## StringTokenizer in Java | Use, Example

**StringTokenizer in Java** is a utility class that breaks a string into pieces or multiple strings called "tokens".

For example, strings separated by space and tab characters are tokens. These tokens are separated with the help of a delimiter.

A Delimiter is a group of characters that separate tokens. Each character in the delimiter is considered a valid delimiter. For example, comma, semicolons, and colon are sets of delimiters.

By default, space, tab, newline, and carriage return are used to separate the strings. But we can also use any other characters to separate tokens.

Let's understand Java StringTokenizer with an example.

Suppose we have a string "Hello Scientech Easy". If we define a space as a delimiter, this string has the following three tokens:

1. Hello
2. Scientech
3. Easy

If we define a comma as a delimiter after "Hello", the same string has the following two tokens:

1. Hello
2. Scientech Easy

StringTokenizer was added in Java 1.0 version and now it is not part of the Java Collections Framework. It is present in java.util package.

If you want to use StringTokenizer, you must import java.util package or at least StringTokenizer class into your application.

## StringTokenizer class declaration

---

StringTokenizer class can be declared in a general form as follows:

```
public class StringTokenizer

  extends Object
```

implements Enumeration<Object>

StringTokenizer class implements Java Enumeration interface and extends Object class.

## Constructors of StringTokenizer class

The StringTokenizer class defines the following constructors that are as follows:

**1. StringTokenizer(String str):** This constructor is used to construct a string tokenizer for the specified string str. The parameter str is a string that will be tokenized. The default delimiters are used.
**2. StringTokenizer(String str, String delim):** This constructor is used to construct a string tokenizer for the specified string and delimiter. The parameter delim is a string that represents delimiters to separate the tokens.
**3. StringTokenizer(String str, String delim, boolean delimAsToken):** This form of constructor constructs StringTokenizer with specified string, delimiter, and returnValue.
If the delimAsToken is true, delimiter characters are returned as tokens. If it is false, delimiter characters are not returned and serve to separate tokens. Delimiters are not returned as tokens by the first two forms of the constructor.

## How to use StringTokenizer in Java?

StringTokenizer class in Java is useful to separate tokens. These tokens are then stored in the StringTokenizer object from where they can be retrieved.

To use StringTokenizer class in java, we need to create an object of StringTokenizer class is as follows:

```
StringTokenizer st = new StringTokenizer(str, "delimiter");
```

In the above statement, the actual string str is broken into multiple strings or tokens at the position marked by delimiters. A StringTokenizer object returns one token at a time. We can also change the delimiter anytime.

For example, to break the string “Hello Scientech easy” whenever a comma is found, we can write as follows:

```
StringTokenizer st = new StringTokenizer("Hello Scientech Easy", ",");
```

Similarly, to break a string whenever a comma, or semi-colon, or both are found, we can specify delimiters as follows:

```
String delimiters = ",;";

StringTokenizer st = new StringTokenizer("Hello Scientech Easy", delimiters);
```

Similarly, to break a string with default delimiters such as space, tab, or newline, we can create object as:

```
StringTokenizer st = new StringTokenizer("Hello Scientech Easy");
```

## Methods of StringTokenizer in Java

---

StringTokenizer class has defined the following methods that are as follows:

**1. int countTokens():** This method counts and returns the number of tokens available in the StringTokenizer object.

**2. boolean hasMoreTokens():** This method checks if there are more tokens available in the StringTokenizer object or not. If the next token is available, it will return true.
**3. String nextToken():** This method returns the next token from the string tokenizer object.
**4. String nextToken(String delim):** It returns the next token from the string tokenizer object based on the delimiter.
**5. boolean hasMoreElements():** It returns the same value as the hasMoreTokens method.
**6. Object nextElement():** It returns the same value as the nextToken method, but its return type is Object.

## Java StringTokenizer Example Programs

1. Let's write a program where we will break a string into tokens using StringTokenizer. We will use hasMoreTokens() method to check if we have more tokens and nextToken() method to retrieve the next token from the string.

Look at the program source code to understand better.

**Program source code 1:**

```
import java.util.StringTokenizer;

public class StringTokens {

public static void main(String[] args)

{

// Take a string.

  String str = "He is a gentle man";
// Take a string for delimiters.

   String delimiters = " ,"; // Here, delimiters are a space and a comma.



// Create an object of string tokenizer and break into tokens. Here, delimiters are a space
and a comma.

   StringTokenizer st = new StringTokenizer(str, delimiters);



// Counts the number of tokens available in string tokenizer object.

   int counts = st.countTokens();

   System.out.println("Number of tokens: " +counts);



// Now retrieves tokens from st and display them.

   System.out.println("Tokens are: ");

   while(st.hasMoreTokens()){

               String tokens = st.nextToken();

               System.out.println(tokens);

   }

 }

}
```

Output:

```
Number of tokens: 5

Tokens are:

He

is

a

gentle

man
```

**Explanation:** In this example program, we take a string and break it into tokens whenever a space and comma are found in the string. The tokens are retrieved from string tokenizer object as string and then displayed on the console. Look at the below figure.

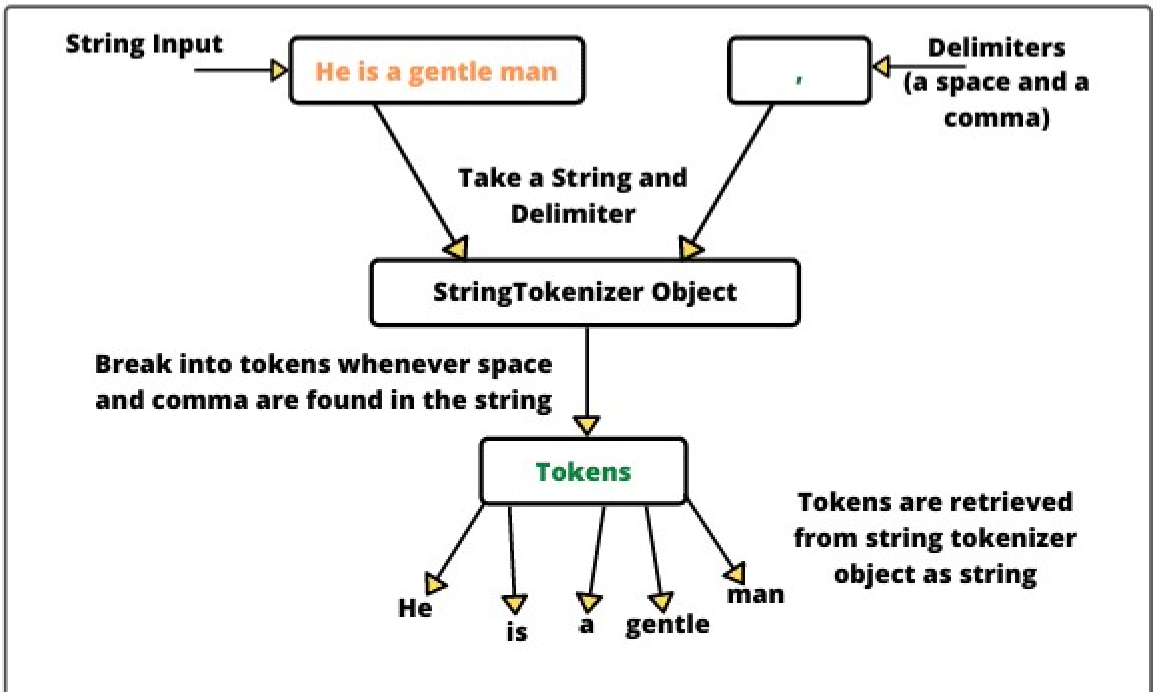

String Input
He is a gentle man
,
Delimiters (a space and a comma)
Take a String and Delimiter
StringTokenizer Object
Break into tokens whenever space and comma are found in the string
Tokens
Tokens are retrieved from string tokenizer object as string
He
is
a
gentle
man

Fig: How StringTokenizer works in Java

2. Let's take another example program where we will split a string into tokens based on delimiters using split() method of String class.

The split() method of String class takes a regular expression as a delimiter. Look at the source code.

**Program source code 2:**

```
public class StringTokens {

public static void main(String[] args)

{

// Take a string.

  String str = "You are very sweet girl";



// Take a string for delimiters.

   String delimiters = "[ ,]+"; // Here, delimiters are a space and a comma.



// Split the string into tokens using String.split() method.

   System.out.println("Tokens are: ");

   String[] s = str.split(delimiters);



   for(int i = 0 ; i < s.length; i++) {

                System.out.println(s[i]);

   }

 }

}
```

```
Output:
   Tokens are:
   You
   are
   very
   sweet
   girl
```

**Explanation:**
As you can see in the output, spilt() method of String class breaks the string tokens and returns each token as a string.

## Java Calendar Class | Methods, Example

**Calendar class in Java** is an abstract super class that is designed to support different kinds of calendar systems. It is used to know system data and time information.
Java Calendar class provides some specific functionality that is common to most calendar systems and makes that functionality available to the subclasses.

The subclasses of Calendar class are used to calculate calendar related information (such as second, minute, hour, day of month, month, year, and so on) of a particular calendar system.

For example, a GregorianCalendar is a subclass of the Java Calendar class that is used to determine the day of week of January 15, 2021, in the Gregorian calendar system.

# Java Map Interface Implementation Classes

The calendar class was introduced in Java 1.1 version. The calendar API is present in java.util.Calendar.

Java 1.4 version introduced the Thai Buddhist Calendar that is a subclass of calendar class. In Java 6, JapaneseImperialCalendar was introduced which is also a subclass Calendar.

## Java Calendar class declaration

---

The abstract Calendar class extends Object class and implements Serializable, Cloneable, and Comparable interfaces. The general declaration of calendar class is as follows:

```
public abstract class Calendar

  extends Object

    implements Serializable, Cloneable, Comparable<Calendar>
```

## Constructors of Java Calendar class

---

Calendar class in Java does not provide any public constructor. It provides constructors with protected access modifier that cannot be used outside the package. They are:

**1. protected Calendar():** This form of constructor is used by subclass to construct a Calendar with the default time zone and locale.
**2. protected Calendar(TimeZone zone, Locale aLocale):** This form of constructor is used by subclass to construct a calendar with the specified time zone and locale.

## How to make/create a Calendar in Java?

To create an object of Calendar class in java, we need to call getInstance() method. Since this method is static in nature so we call it using Calendar class name like this:

```
Calendar cl = Calendar.getInstance();
```

This Calendar object stores the current system date and time by default.

## Methods of Calendar class in Java

Calendar class provides a set of methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. Some commonly used methods defined by Calendar are as follows:

**1. abstract void add(int field, int amount):** This method is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.
**2. boolean after(Object calendarObj):** This method returns true if the invoking calendar object contains a date that is after (later) than specified calendarObj. Otherwise, it returns false.
**3. boolean before(Object calendarObj):** This method returns true if the invoking calendar object contains a date that is before (earlier) than specified calendarObj. Otherwise, it returns false.
**4. final void clear():** It sets zeros all time components in the invoking calendar object.
**5. final void clear(int field):** It sets zeros the specified time components in the invoking calendar object.

**6. Object clone():** It creates and returns a duplicate copy of invoking object.
**7. int compareTo(Calendar anotherCalendar):** It is used to compare the time values (milliseconds offsets) between two Calendar objects.
**8. boolean equals(Object calendarObj):** It is used to compare between invoking calendar object and the specified Object calendarObj. It returns true if they are the same. Otherwise, returns false.
**9. protected void complete():** This method fills in any unset fields in the calendar fields.
**10. protected abstract void computeFields():** This method converts the current millisecond time value time to calendar field values in fields[ ].

---

**11. protected abstract void computeTime():** This method is used to convert the current calendar field values in fields[ ] to the millisecond time value time.
**12. int get(int calendarField):** It returns the value of the given calendar field.
**13. int getActualMaximum(int calendarField):** It is used to get the actual maximum value of the calendar field passed as the parameter to getActualMaximum() method.
**14. int getActualMinimum(int calendarField):** It is used to get the actual minimum value of the calendar field passed as the parameter to getActualMaximum() method.
**15. static Locale[ ] getAvailableLocales():** It returns an array of available Locale objects that contains locales for which calendars are available.

---

**16. String getDisplayName(int field, int style, Locale locale):** It returns the string representation of the calendar field value in the given style and locale.
**17. int getFirstDayOfWeek():** It is used to get the first day of the week integer form. For example, SUNDAY in India., MONDAY in USA.
**18. static Calendar getInstance():** It is used to get a calendar object for the default time zone and locale.
**19. static Calendar getInstance(Locale locale):** It is used to get the calendar object with the specified locale and default time zone.
**20. static Calendar getInstance(TimeZone zone):** It is used to get a calendar object with the specified time zone and default locale.

---

**21. static Calendar getInstance(TimeZone zone, Locale locale):** It returns a calendar object with the specified time zone and locale.
**22. final Date getTime():** It returns a Date object that is equal to the time of invoking object.
**23. long getTimeInMillis():** It returns time value of calendar in milliseconds.
**24. TimeZone getTimeZone():** It is used to get the time zone for the invoking object.
**25. int getWeeksInWeekYear():** It is used to get the total number of weeks in the week year represented by the Calendar.

---

**26. int getWeekYear():** It returns the week year represented by the Calendar.
**27. int hashCode():** It returns the hash code value for the calendar.
**28. final boolean isSet(int field):** It returns true if the specified time component of the given calendar is set otherwise, returns false.
**29. boolean isWeekDateSupported():** It returns true if the Calendar supports week dates.

**30. void set(int field, int value):** It sets the date or time component specified by calendar field to the given value in the invoking object.

---

**31. void set(int year, int month, int date):** It sets year, month, and day_of_month for a given calendar.
**32. void set(int year, int month, int date, int hourOfDay, int minute):** It sets the values for the calendar fields such as year, month, day, hour, and minute.
**33. void set(int year, int month, int date, int hourOfDay, int minute, int second):** It sets the values for the calendar fields like year, month, day, hour, minute, and second.
**34. void setFirstDayOfWeek(int value):** It sets the first day of the week for a given calendar.
**35. void setTime(Date date):** It sets the time for the specified calendar object with the given Date.

---

**36. void setTimeInMillis(long millis):** It sets the current time in milliseconds for the given calendar object.
**37. void setTimeZone(TimeZone value):** It sets the time zone for invoking object with the given time zone value.
**38. String toString():** It returns a string representation of the given calendar object.

## Calendar Field Constants

---

Calendar class defines the following int constants that are used to get or set components of the calendar. They are listed in the table.

# Java Map Interface Implementation Classes

| | | | |
|---|---|---|---|
| ALL_STYLES | FRIDAY | PM | AM |
| HOUR | SATURDAY | AM_PM | HOUR_OF_DAY |
| SECOND | APRIL | JANUARY | SEPTEMBER |
| AUGUST | JULY | SHORT | DATE |
| JUNE | SUNDAY | DAY_OF_MONTH | LONG |
| THURSDAY | DAY_OF_WEEK | MARCH | TUESDAY |
| DAY_OF_WEEK_IN_MONTH | MAY | UNDECIMBER | DAY_OF_YEAR |
| MILLISECOND | WEDNESDAY | DECEMBER | MINUTE |
| WEEK_OF_MONTH | DST_OFFSET | MONDAY | WEEK_OF_YEAR |
| ERA | MONTH | YEAR | FEBRUARY |
| NOVEMBER | ZONE_OFFSET | FIELD_COUNT | OCTOBER |

## Instance Variables of Calendar

Java Calendar class defines several protected instance variables that are as follows:

1. areFieldsSet: It is a variable of type boolean that indicates if the time components have been set.

2. fields: It is an array of type ints that stores the components of the time.

3. isSet: It is a variable of type boolean array that indicates if a specific time component has been set.

4. time: It is a variable of type long that stores the current time for the object.

5. isTimeSet: It is a boolean variable that indicates if the current time has been set.

## Java Calendar Example Programs

---

Let's take various useful example programs based on the java calendar.

1. Let's create a program where we will display the system date and time. Look at the program source code to understand better.

**Program source code 1:**

```
import java.util.Calendar;

public class CalendarDemo {

public static void main(String[] args)

{

// Create a calendar class object and initialized with the current date and time in the
default locale and timezone.

   Calendar cl = Calendar.getInstance();



// Display current date information separately.

    System.out.print("Current System Date: ");
```

```
  int dd = cl.get(Calendar.DATE);

  int mm = cl.get(Calendar.MONTH);



  ++mm;

  int yy = cl.get(Calendar.YEAR);

  System.out.println(dd+ "-" +mm+ "-" +yy);



// Display time information.

  System.out.print("Current System Time: ");



  int hr = cl.get(Calendar.HOUR);

  int min = cl.get(Calendar.MINUTE);

  int sec = cl.get(Calendar.SECOND);

  int secMilli = cl.get(Calendar.MILLISECOND);



  System.out.println(hr+ ":"+min+ ":"+ sec+ ":"+secMilli);



// Call getTime() method to get the present date and time information.

    System.out.println("At present Date and Time: " +cl.getTime());

 }
```

```
}

Output:

     Current System Date: 19-1-2021

     Current System Time: 3:56:35:343

     At present Date and Time: Wed Jan 20 08:35:28 IST 2021
```

**Explanation:** In this example program, we create an object of Calendar class that contains by default data and time information as shown by the local system.
From the calendar object, we get the date and time information separately using get() method and print them on the console.

2. Let’s create a program where we will get maximum and minimum numbers of days in a week, maximum and minimum number of weeks in a year. Look at the source code.

**Program source code 2:**

```
import java.util.Calendar;

public class CalendarDemo {

public static void main(String[] args)

{

// Create a calendar class object.

  Calendar cl = Calendar.getInstance();



  int maximum = cl.getMaximum(Calendar.DAY_OF_WEEK);

  System.out.println("Maximum number of days in week: " + maximum);
  maximum = cl.getMaximum(Calendar.WEEK_OF_YEAR);

  System.out.println("Maximum number of weeks in year: " + maximum);



  int minimum = cl.getMinimum(Calendar.DAY_OF_WEEK);

  System.out.println("Minimum number of days in week: " + minimum);



  minimum = cl.getMinimum(Calendar.WEEK_OF_YEAR);

  System.out.println("Minimum number of weeks in year: " + minimum);

 }

}
```

Output:

```
    Maximum number of days in week: 7

    Maximum number of weeks in year: 53

    Minimum number of days in week: 1

    Minimum number of weeks in year: 1
```

3. Let’s take an example program and we will know the current date and time information, 20 days ago, 5 months later, and 5 years later date and time information from present now. Look at the source code.

**Program source code 3:**

```
import java.util.Calendar;
```

```
public class CalendarDemo {

public static void main(String[] args)

{

// Create a calendar class object.

   Calendar cl = Calendar.getInstance();



 System.out.println("Current Date: " + cl.getTime());



 cl.add(Calendar.DATE, -20);

 System.out.println("20 Days ago, date and time information: " + cl.getTime());



 cl.add(Calendar.MONTH, 5);

 System.out.println("5 months later, date and time information: " + cl.getTime());



 cl.add(Calendar.YEAR, 5);

 System.out.println("5 years later, date and time information: " + cl.getTime());

  }

}

Output:

     Current Date: Wed Jan 20 09:01:44 IST 2021
```

```
20 Days ago, date and time information: Thu Dec 31 09:01:44 IST 2020

5 months later, date and time information: Mon May 31 09:01:44 IST 2021

5 years later, date and time information: Sun May 31 09:01:44 IST 2026
```

## How to use Calendar in Java?

There are two ways to use the Calendar class in Java. They are as follows:

1. Calendar class helps to know the system date and time information.
2. It helps to store a date and time value so that it can be carried to some other applications.

## Java GregorianCalendar | Example Program

**GregorianCalendar in Java** is a concrete subclass of an abstract class Calender.
In other words, it is an implementation of a Calendar class in Java API. It is the most commonly used calendar.

A calendar in java is an abstract superclass for obtaining detailed calendar information such as year, month, date, hour, minute, and second.
Since Calendar class is an abstract class, we cannot instantiate it. That is, we cannot create an object of an abstract class. But we can create an object of GregorianCalendar because it is a concrete class.

GregorianCalendar was introduced in JDK 1.1. It is present in java.util.GregorianCalendar package.

### GregorianCalendar class declaration

## Java Map Interface Implementation Classes

GregorianCalendar is a concrete class that can be declared as follows:

```
public class GregorianCalendar extends Calendar
```

Java GregorianCalendar class extends Calendar class and implements Serializable, Cloneable, and Comparable<Calendar> interfaces. Calendar class extends Object class.

### GregorianCalendar Class Constructors in Java

Java GregorianCalendar class provides several constructors that are as follows:

**1. GregorianCalendar():** This constructor creates a default GregorianCalendar using the current time in the default time zone with the default locale.
In other words, it initializes the object with the current date and time in the default locale and time zone.

The syntax to create an object of GregorianCalendar class for current data and time is as follows:

```
GregorianCalendar gcal = new GregorianCalendar();
```

**2. GregorianCalendar(int year, int month, int dayOfMonth):** This form of constructor creates a GregorianCalendar for the specified year, month, and date.
The general syntax to create an object of GregorianCalendar for the specified year, month, and date is as follows:

```
GregorianCalendar gcal = new GregorianCalendar(int year, int month, int dayOfMonth);
```

```
For example:

GregorianCalendar gcal = new GregorianCalendar(2021, 2, 10);
```

**3. GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute):** This form of constructor creates a GregorianCalendar for the specified year, month, date, hour, and minute. The general syntax to create an object of GregorianCalendar for the specified year, month, date, hour, and minute is as follows:

```
GregorianCalendar gcal = new GregorianCalendar(int year, int month, int dayOfMonth,
int hourOfDay, int minute);



For example:

GregorianCalendar gcal = new GregorianCalendar(2021, 2, 10, 09, 50);
```

**4. GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute, int second):** This constructor constructs a GregorianCalendar for the specified year, month, date, hour, minute, and second.
The general syntax to create an object of GregorianCalendar for the specified year, month, date, hour, minute, and second is as follows:

```
GregorianCalendar gcal = new GregorianCalendar(int year, int month, int dayOfMonth,
int hourOfDay, int minute, int second);



For example:

GregorianCalendar gcal = new GregorianCalendar(2021, 2, 10, 09, 50, 25);
```

**5. GregorianCalendar(Locale aLocale):** It creates a GregorianCalendar based on the current time in the default time zone with the specified locale.
**6. GregorianCalendar(TimeZone zone):** This constructor constructs a GregorianCalendar based on the current time in the specified time zone and default locale.

**7. GregorianCalendar(TimeZone zone, Locale aLocale):** It creates a GregorianCalendar based on the current time in the specified time zone and locale.

## GregorianCalendar Methods in Java

---

In addition to methods inherited from Calendar class and Object class, Java GregorianCalendar class also provide some useful methods. They are as follows:

**1. void add(int field, int amount):** This method is used to add the specified (signed) amount of time to the specified calendar field, based on the calendar's rules.
**2. Object clone():** This method creates and returns a copy of this object.
**3. protected void computeFields():** This method converts the time value to calendar field values.
**4. protected void computeTime():** It converts calendar field values to the time value.
**5. boolean equals(Object obj):** This method is used to compare the GregorianCalendar object to the specified Object obj.

---

**6. int getActualMaximum(int field):** This method is used to retrieve the maximum value that a specified calendar field can have.

**7. int getActualMinimum(int field):** This method is used to retrieve the minimum value that a specified calendar field can have.
**8. int getGreatestMinimum(int field):** It retrieves the highest minimum value for the specified calendar field of the current GregorianCalendar instance.
**9. Date getGregorianChange():** It gets the Gregorian Calendar change date.
**10. int getLeastMaximum(int field):** This method returns the lowest maximum value for the specified calendar field of the current GregorianCalendar instance.

---

**11. int getMaximum(int field):** It returns the maximum value for the specified calendar field of the current GregorianCalendar instance.
**12. int getMinimum(int field):** It returns the minimum value for the specified calendar field of the current GregorianCalendar instance.
**13. TimeZone getTimeZone():** It is used to get the time zone.
**14. int getWeeksInWeekYear():** It returns the number of weeks in the week year represented by the GregorianCalendar.
**15. int getWeekYear():** This method returns the week year represented by the GregorianCalendar.

---

**16. int hashCode():** It generates the hash code value for the GregorianCalendar object.
**17. boolean isLeapYear(int year):** The isLeapYear() method is used to determine if the specified year is a leap year or not. It returns true if year is a leap year otherwise, returns false.
**18. boolean isWeekDateSupported():** It returns true if the GregorianCalendar supports week dates.

**19. void roll(int field, boolean up):** It adds or subtracts (up/down) a single unit of time on the specified time field without changing larger fields.
**20. void roll(int field, int amount):** It adds a signed amount to the given calendar field without changing larger fields.

---

**21. void setGregorianChange(Date date):** It sets the GregorianCalendar change date.
**22. void setTimeZone(TimeZone zone):** It sets the time zone with the specified time zone value.

---

**Note:** Java GregorianCalendar class also inherits several constant fields and methods from the Calendar class. These contact fields are used as arguments to methods in the class.

## Example Program based on GregorianCalendar in Java

---

1. Let's take an example program where we will display the current date and time information of the system. We will also check that the current year is a leap year or not using isYearLeap() method provided by Java GregorianCalendar.

**Program source code 1:**

```
import java.util.Calendar;

import java.util.GregorianCalendar;

public final class GCalendarTest {
```

# Java Map Interface Implementation Classes

```
public static void main(String[] args)

{

//  Create an array of type String to store month abbreviations.

String months[ ] = {

        "Jan", "Feb", "Mar", "Apr",

         "May", "Jun", "Jul", "Aug",

                              "Sep", "Oct", "Nov", "Dec"};

 int year;



// Create an object of Gregorian calendar initialized with the current date and time in the
default locale and timezone.

     GregorianCalendar gcal = new GregorianCalendar();



// Display current date and time information.

    System.out.print("Date Info: ");

    System.out.print(months[gcal.get(Calendar.MONTH)]);



    System.out.print(" " + gcal.get(Calendar.DATE) + ", ");

    System.out.println(year = gcal.get(Calendar.YEAR));
```

```
  System.out.print("Time Info: ");

  System.out.print(gcal.get(Calendar.HOUR) + ":");



  System.out.print(gcal.get(Calendar.MINUTE) + ":");

  System.out.println(gcal.get(Calendar.SECOND));



// Test if the current year is a leap year or not.

   if(gcal.isLeapYear(year))

   {

        System.out.println("Current year is a leap year");

   }

    else {

                System.out.println("Current year is not a leap year");

    }

   }

}

Output:

      Date: Feb 9, 2021

      Time: 0:43:14

      Current year is not a leap year
```

## Java Map Interface Implementation Classes

2. Let's create a calendar for February 2021 and will determine the name of the day of month. For this, we will create an object of Gregorian calendar class and pass year, month, and day as arguments to its constructor.

Look at the program source code given below.

**Program source code 2:**

```
import java.util.Calendar;

import java.util.GregorianCalendar;

public final class Test {

public static void main(String[] args)

{

//  Create an array of type String to store name of days.

String dayNameOfWeek[] = {

        "Sunday", "Monday", "Tuesday", "Wednesday",

         "Thursday", "Friday", "Saturday"};



// Create an object of Gregorian calendar initialized with February 9 2021.

   GregorianCalendar gcal = new GregorianCalendar(2021, 2, 9); // Passing year, month,
dayOfMonth as parameters.



  System.out.println("February 09, 2021, is a " +
dayNameOfWeek[gcal.get(Calendar.DAY_OF_WEEK) - 1]);

  }
}

Output:

     February 09, 2021, is a Tuesday
```